QFLOW: DESARROLLO DE APLICACIONES PARA LA GESTIÓN DE COLAS

QFLOW: APPS DEVELOPMENT FOR MANAGING QUEUES

Rubén Izquierdo Belinchón

Rocío García Núñez

Víctor Gómez – Jareño Guerrero

Daniel Piña Miguelsanz

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO

CURSO 2019-2020

Directoras: DRA. Victoria López López, DRA. Matilde Santos Peñas

AGRADECIMIENTOS

A nuestras familias y amigos, por todo el apoyo que nos han mostrado durante esta etapa tan importante en nuestras vidas. En especial a Cris por ser un gran punto de apoyo y ofrecernos su ayuda cuando la necesitamos, también a Kiko, Ángeles, Ana, Felipe, Charo, Carmen y Manuel por estar ahí en los momentos más difíciles.

A nuestras tutoras Victoria López López y Matilde Santos Peñas por darnos la oportunidad de desarrollar este proyecto.

RESUMEN

Los espacios ocupados por las colas presenciales suponen una problemática en la actualidad que se plantea solucionar a partir de este Trabajo Fin de Grado. Para ello, se ha realizado un estudio de mercado de las alternativas actuales, además de la aplicación de una base matemática sobre teoría de colas y una utilización de metodologías ágiles, concretamente Scrum.

Como resultado del proyecto se han generado dos aplicaciones, una orientada para usuarios y otra para creadores. Ambas han sido desarrolladas para Android, utilizando como lenguaje de programación Kotlin, debido a sus ventajas sobre Java. Para su implementación, se ha aplicado la Arquitectura Clean, junto al patrón *Model View ViewModel*, además de añadir herramientas relacionadas con la conexión con los servidores, lecturas de códigos QR, sistemas de persistencia encriptada e inyección de dependencias.

Estas aplicaciones quedan respaldadas por un servidor desarrollado en Spring Boot, que aplica la misma arquitectura. Su implementación se ha realizado siguiendo las buenas prácticas del desarrollo guiado por test, y su despliegue ha sido llevado a cabo mediante la plataforma Heroku, aportando también una documentación realizada gracias a la herramienta Swagger.

Palabras clave

Teoría de colas, Kotlin, PostGreSQL, Android, Arquitectura Clean, Retrofit, Spring, Koin, Swagger, Flavours, Flyway y Heroku.

ABSTRACT

In this End of Grade Work a solution arises to the problem of the spaces occupied by the face-to-face queues of today. To this end, a market research of current alternatives, a mathematical basis on queue theory and the use of agile methodologies, specifically Scrum, have been applied.

We have developed two applications, one user-oriented and one for creators. Both have been developed for Android, using Kotlin as the main programming language, thanks to its advantages over Java. For their implementation, the Clean Architecture was applied, together with the Model View ViewModel pattern. In addition to adding tools related to server connection, QR code reading, encrypted persistence systems and dependency injection.

These applications are backed by a server developed in Spring Boot, which applies the same architecture. Its implementation has been carried out following the best practices of test-driven development, and its deployment has been done through the Heroku platform. Documentation is also provided using the Swagger tool.

Keywords

Queueing theory, Kotlin, PostGreSQL, Android, Clean Architecture, Retrofit, Spring, Koin, Swagger, Flavours, Flyway and Heroku.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 - Introducción

Nuestra motivación principal para desarrollar este proyecto ha sido solventar la problemática que tienen los usuarios cuando se encuentran en largas colas de espera en las cuales tienen que permanecer de pie sin poder invertir su tiempo de manera más eficaz. Al ver que no existía una solución generalizada para superar estas colas de espera surgió la idea de crear está aplicación para móvil.

Para acceder al repositorio donde están alojadas las aplicaciones de Android, clic en el siguiente enlace: QFlow, y para acceder al repositorio donde está el server, consulta el siguiente enlace: QFlowServer.

1.1 Objetivos

El objetivo del proyecto es desarrollar un *framework* que permita optimizar las esperas en las colas de los grandes eventos con gran afluencia de gente. Para esto se creará una plataforma donde, por un lado, los asistentes puedan conocer en todo momento información sobre su situación en la lista de espera, y por otro se les dé a los creadores de estas colas la posibilidad de gestionarlas basándonos en los fundamentos de la teoría de colas.

1.1.1 Asignaturas

Se ha analizado la relación con ciertas asignaturas que se han cursado durante la carrera con el fin de llevar este objetivo a cabo.

Gracias a *Auditoría Informática* se ha podido organizar la memoria de manera óptima. Para implementar la parte de colas han sido de gran ayuda las asignaturas de *Evaluación de Configuraciones, Probabilidad y Estadística* y *Sistemas Operativos.* En el diseño de las aplicaciones, las asignaturas *Aplicaciones Web* y *Desarrollo de Sistemas Interactivos*, junto a la base de Java obtenida en *Tecnologías de la Programación*. Por último, para poder organizar el proyecto de la manera más eficaz se han aplicado los conocimientos adquiridos en *Ingeniería del Software* sobre los patrones de diseño software.

Aunque estas son las asignaturas principales, en general, otras muchas asignaturas de la carrera han dado la base necesaria para poder desarrollar y llevar a término este trabajo.

1.1.2 Metodología

Para el desarrollo del proyecto se ha decidido adoptar una metodología ágil como es Scrum [1], que divide el desarrollo en diferentes *sprints*, con tiempo fijo, repetible, durante el cual se crea un producto del menor valor posible. Una descripción sencilla de los pasos a seguir sería la siguiente:

Al principio de cada uno de los *sprints* se define el *product backlog* o tareas a realizar durante el *sprint*. Posteriormente, se realiza un *Sprint Planning* en el cual se transforman objetos definidos en el *product backlog* en las tareas a realizar durante el sprint. Una vez completado este paso, se añaden en el *Sprint Backlog*.

Con las tareas definidas para la semana siguiente se realizan dichas actividades. Para una aplicación correcta de la metodología se debería realizar una reunión diaria conocida como *Daily Scrum*, pero en nuestro caso únicamente se le ha dado uso durante los *sprints* más intensos de desarrollo.

Una vez terminada la semana, se analiza el *sprint* en el *Sprint Review*, y volvemos a comenzar el ciclo.

Dentro de esta metodología existe una figura conocida como *Scrum Master*, el cual es un integrante del equipo que queda definido por el propio grupo para encargarse de asesorar en la organización del proyecto, moderar las reuniones, resolver impedimentos en los *sprint* y asegurarse del cumplimiento de la metodología. En nuestro caso, para favorecer el aprendizaje, esta posición ha ido rotando por *sprint*, de manera que todos hemos tenido la oportunidad de practicar ese puesto.

Lo habitual en Scrum es la utilización de un *Kanban* para repartir las tareas y organizar el equipo. Por ello, se ha utilizado una aplicación web conocida como Trello que se comentará en la sección de tecnologías utilizadas.

Para establecer un buen control en la gestión de versiones se ha utilizado la herramienta GitHub. Por otro lado, el flujo de trabajo elegido es el denominado GitFlow. Éste se organiza en dos ramas principales: *master* y *dev*, junto con ramas auxiliares como *feature* o *hotfix*. La rama *master* nunca va a ir por delante de la rama *dev* excepto cuando se haya terminado una fase del código y se suba a la rama *dev* y a la master. En *dev* se introducen las funcionalidades completadas. Cada una de éstas se realiza en una subrama *feature*. De esta forma se controlan las distintas

versiones evitando colisión entre ellas. Para lanzar una nueva versión se entrega al cliente una versión de *release*, es otra rama que actualiza el número de versión de la aplicación partiendo de *master*.

1.2 Estructura de la memoria

Esta memoria se ha organizado de la siguiente forma:

- En este primer capítulo se introduce el proyecto, las asignaturas que han sido importantes para realizarlo y la metodología de desarrollo que se ha seguido.
- En el segundo capítulo se comenta el estado del arte en el comienzo del desarrollo del proyecto, un estudio de aplicaciones similares y un apartado explicando la teoría de colas.
- En el tercer capítulo se detallan las herramientas usadas durante el desarrollo del proyecto.
- En el cuarto capítulo se explican los casos de uso implementados en las aplicaciones.
- El quinto capítulo detalla los servicios web y la base de datos usada para desarrollar las aplicaciones.
- El sexto capítulo describe toda la arquitectura de las aplicaciones, la estructura de estas y todas las funcionalidades implementadas en las aplicaciones.
- El capítulo 7 explica el servicio usado y las funcionalidades del servidor.
- En el capítulo 8 se presentan las conclusiones y el trabajo futuro desarrollable a partir de la finalización del proyecto.
- Por último, se han añadido cuatro anexos, el primero es una guía de usuario para el uso de las apks, el segundo explica la arquitectura de Android, el tercero cuenta como se hizo el despliegue de Heroku y el último explica las distribuciones de tipo discreto.

Capítulo 2 - Estado del arte

Para obtener una mejor comprensión del proyecto, se ha realizado una investigación de aplicaciones con objetivos similares a la nuestra. Así, se ha podido obtener ideas sobre qué mejorar en nuestra aplicación al mismo tiempo que se ha podido contemplar y conocer la competencia actual.

2.1 Análisis de las aplicaciones

En esta sección se presentan los aspectos más relevantes de cada aplicación, sus ventajas y desventajas, y lo que nos diferenciará de estas. En el apartado 2.1.5 se muestra la Tabla 2.1 que resume esta comparativa.

2.1.1 Skiplino

Skiplino [2] es una aplicación de gestor de colas, que ofrece soluciones para mejorar el flujo de tráfico y ahorrar tiempo a las personas. Es decir, persigue nuestro mismo objetivo.

Consta de más de 10.000 descargas en la Play Store. Fue actualizado por última vez el 2 de diciembre de 2019. *Skiplino* es la aplicación con mejor diseño de las que se han comparado, que busca ahorrar tiempo a sus clientes. Ésta se divide en realidad en 3 aplicaciones diferentes, según los distintos clientes que le van a dar uso. La principal es *Skiplino*, en la que los usuarios buscan ahorrarse la espera en diferentes colas, situadas en un directorio. *Skiplino Admin* se dirige a los establecimientos que deseen poner en uso el sistema de *Skiplino*; éste ofrece amplias estadísticas como ocupación de colas, opinión de clientes, etc., así como monitorización y gestión de colas del negocio en directo. Por último, *Skiplino Branch*, que se divide en otras dos aplicaciones para Android, una normal y otra para televisores donde un cliente se podrá registrar en la cola sin tener que acceder a su dispositivo móvil personal, y ver el número y ocupación de la cola en los televisores, con *Branch TV*.

Entre las capacidades que listan en la Play Store se destaca el acceso a colas de un gran número de proveedores, el acceso remoto, el directorio de colas por tipos de establecimiento y el sistema de notificaciones para informar de que el usuario llega tarde.

Otros aspectos diferenciales son su diseño actualizado y minimalista, tal como se muestra en la Figura 2.1, la multitud de aplicaciones diferentes para cubrir todas las necesidades de un cliente o proveedor, y el servicio de notificaciones que se presenta ante el usuario. Eso sí, no está en funcionamiento en Europa y únicamente está en uso en el Oriente Medio.

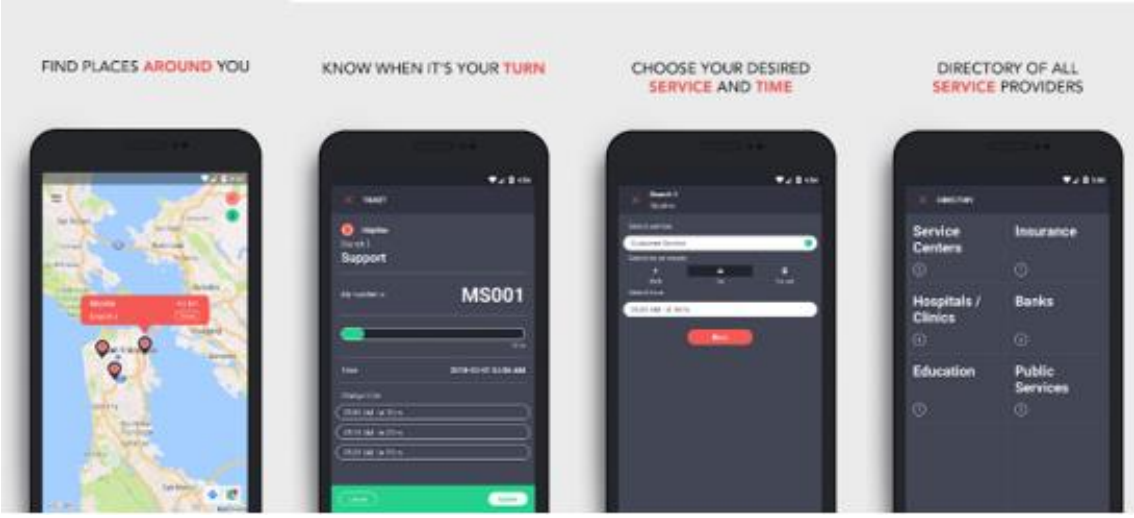


Figura 2.1 Funcionalidades de Skiplino

2.1.2 QLess

QLess [3] es una aplicación que permite esperar una cola virtual en vez de una física. Al unirse a ella, los clientes reciben un mensaje de texto, una llamada y una notificación cuando se acercan a la cola. Es otra aplicación con nuestro mismo objetivo y, en el momento de realizar el proyecto, es la que más descargas tenía en la Play Store.

Con más de 100.000 descargas se consolida como una aplicación a tener en cuenta. Su última actualización fue el 15 de mayo de 2019. Consta de una nota media de 4,4 estrellas y más de 2.000 *reviews*.

Entre sus muchas capacidades destaca la interfaz de un mapa donde se muestran las colas cercanas al usuario, pudiendo así controlar dónde están exactamente las colas a las que el usuario se puede unir. Este puede acceder a un listado de estas para así poder controlar a las que se ha unido e incluso, solicitar más tiempo en la cita que le haya sido concedida (ver Figura 2.2).

Por estas y otras capacidades, el Gobierno de los EE. UU. ha incluido esta aplicación en sus cuerpos administrativos para que el público la pueda usar.

La única desventaja notable es su ausencia en Europa.

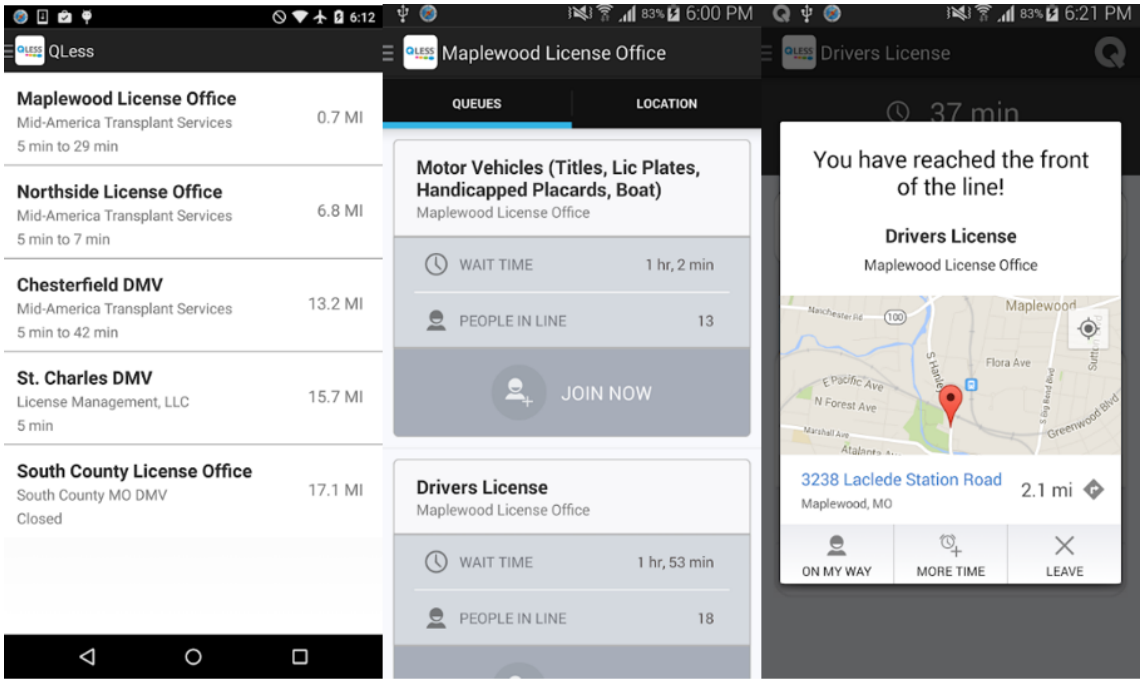


Figura 2.2 Funcionalidades QLess

2.1.3 WhyLine

Es una aplicación muy parecida a *QLess* y a *Skiplino*, ofreciendo las mismas características a los usuarios.

WhyLine [4] consta de más de 100.000 descargas en la Play Store y una nota media de 3.5 estrellas con más de 500 *reviews*. Su última actualización fue lanzada el 27 de noviembre de 2019.

De entre todas las aplicaciones que se han comparado e investigado, *WhyLine* tiene la mejor interfaz. Con un buen diseño, de fácil aprendizaje y sencilla, a la par que mezcla funcionalidades completas y potentes.

Esta aplicación permite al usuario elegir por tipo de servicio, lo que le lleva a un mapa en donde puede elegir las colas a las que quiera unirse, mostrando un tiempo estimado y los usuarios que tiene delante. Aparte de esto, la aplicación le permite retrasar la cita y retroceder posiciones (ver Figura 2.3).

WhyLine ofrece una categoría de sitios favoritos del usuario, dando mayor rapidez en las colas rutinarias. El usuario puede añadir su propia estimación del tiempo de espera y de la ocupación de la cola, con lo que, junto con otras fuentes, *WhyLine* calcula una cifra lo más real posible.

Al igual que *QLess*, *WhyLine* no tiene servicio en Europa, siendo, de nuevo un aliciente para desarrollar nuestra aplicación.

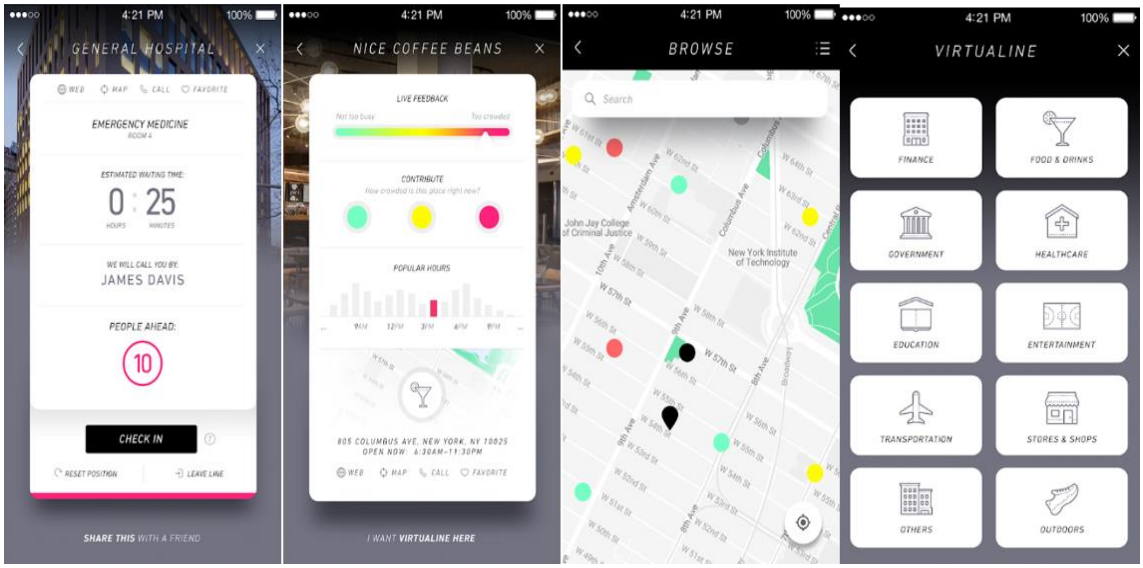


Figura 2.3 Funcionalidades WhyLine

2.1.4 Otras aplicaciones estudiadas

Las aplicaciones *Myfuncwait3* y *Alcampo* no se analizaron con tanto detalle porque se consideró que no contemplaban la misma funcionalidad que se quería implementar en nuestro proyecto. Entre sus funcionalidades se encuentran la de implementar una gamificación para la espera de las colas, funcionalidad a considerar, y la espera de colas exclusivamente dentro de la marca.

2.1.5 Tabla comparativa entre aplicaciones existentes y nuestra aplicación

Tras hacer un estudio más detallado de las aplicaciones que son similares a la nuestra, se decide agrupar toda la información en una tabla comparativa (Tabla 2.1) para tener una idea más general de a qué nos enfrentamos y qué ideas podemos innovar.

Se ha marcado con el símbolo ✓ cuando la aplicación correspondiente sí tiene esa funcionalidad y con una ✗ para indicar lo contrario.

Tabla 2.1. Comparativa de aplicaciones

	Skiplino	QLess	WhyLine	Requisitos de aplicación a realizar
Dispositivos móviles	✓	✓	✓	✓
Aplicación web	✓	✗	✗	✓
Avisos	✓	✓	✓	✓
Ver información sobre la cola	✓	✓	✓	✓
Cola en la que se encuentra el usuario	✓	✓	✓	✓
Mostrar el tiempo de espera en la cola	✓	✓	✓	✓
Mostrar que usuarios han accedido a la cola	✗	✓	✓	✓
Visualiza el tiempo medio de espera en la cola	✗	✓	✓	✓
Estadísticas	✓	✗	✗	✗
Darle permiso al usuario para modificar su tiempo por si llega tarde	✓	✓	✗	✗
Cancelar turno	✓	✗	✗	✗
Geolocalización	✓	✓	✓	✗

2.2 Teoría de colas

En este apartado se detalla en qué consiste la teoría de colas y su posterior aplicación en nuestro proyecto.

2.2.1 Descripción sistemas de colas

La teoría de colas [5] es una rama de la investigación operativa que estudia el comportamiento de los sistemas de atención a los clientes que demandan un servicio, cuando en ocasiones estos tienen que esperar para ser atendidos.

El objetivo de la teoría de colas [6] es encontrar el equilibrio entre el número de clientes que se encuentran en la línea de espera y la cantidad de servidores que satisfagan la demanda del servicio.

2.2.2 Distribuciones probabilísticas

La problemática de colas viene dada por el carácter aleatorio de la llegada de los clientes y de los tiempos de servicio, que se representan mediante las siguientes distribuciones probabilísticas.

La Distribución de Poisson se corresponde con variables aleatorias del tipo 'número de eventos que ocurren en un periodo de tiempo determinado'. Por ejemplo, el número de llegadas a una cola por minuto sigue una distribución de Poisson, siendo lambda (λ) su parámetro, que se estima como el número medio de observaciones (personas) por unidad de tiempo que llegan a la cola a partir de una muestra aleatoria simple.

La Distribución Exponencial se define como el tiempo que transcurre entre dos eventos consecutivos aleatorios. Así, si consideramos el tiempo que transcurre entre dos llegadas consecutivas a una cola, esa variable se distribuye teóricamente según una Exponencial. Los parámetros de ambas distribuciones son inversos y se diferencian en el tipo de variable, siendo discreta en el caso de la distribución de Poisson y continua en la distribución Exponencial.

Por tanto, la teoría de colas seguirá una distribución Exponencial cuando mida tiempos (variable continua) y una distribución de Poisson cuando mida el número de tareas (variable discreta).

Hay que tener en cuenta que las variables aleatorias con distribución Exponencial no existen en la naturaleza. Esto se ha demostrado matemáticamente mediante una propiedad conocida como la falta de memoria de la exponencial. Aplicado a teoría de colas, esta propiedad significa que el hecho de que vengan muchas personas a una cola en un momento dado no condiciona la probabilidad de que vengan más personas, contradiciendo la idea intuitiva de la demanda. La realidad presenta variables con distribuciones que tienen memoria, es decir, la probabilidad a lo largo del tiempo se condiciona al saber que ha habido una gran demanda de la cola o por el contrario ninguna demanda en el tiempo.

En la teoría de colas se usan las siguientes distribuciones continuas:

- **Erlang:** Mide el tiempo entre llegadas a la cola o el tiempo entre salidas de la cola y la distribución.

  Su función distribución viene dada por [7]:

$$f(x) = \frac{(\lambda x)^{(k-1)}}{(k-1)!}\lambda e^{-\lambda x} \qquad (1)$$

  Donde k [8] es un parámetro de forma determina su desviación estándar:

$$\sigma = \frac{1}{k} \qquad (2)$$

  Sabiendo que si k = 1 entonces la distribución de Erlang es igual a la exponencial y si k = ∞, la distribución de Erlang es igual a la distribución degenerada con tiempos constantes.

  Su esperanza viene dada por:

$$\frac{K}{\lambda} \qquad (3)$$

  Y su varianza viene dada por:

$$\frac{K}{\lambda^2} \qquad (4)$$

- **Weibull:** Al igual que la distribución de Erlang, mide el tiempo entre llegadas a la cola o el tiempo entre salidas de la cola y la distribución.

  Su función de distribución es:

$$F(x) = 1 - e^{\left(\frac{x}{\alpha}\right)^{\beta}} \qquad (5)$$

  La función de densidad de este modelo viene dada por [9]:

$$f(x) = \begin{cases} \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1} e^{-\left(\frac{x}{\alpha}\right)^{\beta}} \; si \; x \geq 0 \\ 0 \; si \; x < 0 \end{cases} \qquad (6)$$

  Depende de los parámetros $\alpha > 0$ es un parámetro de escala y $\beta > 0$ es un parámetro de forma.

  La función de distribución viene dada por:

$$\mathrm{F(x)} = 1 - \mathrm{e}^{\left(\frac{\mathrm{x}}{\alpha}\right)^{\beta}} \qquad (7)$$

  La esperanza es:

$$\mathrm{E[X]} = \alpha\Gamma\left(\frac{1}{\beta} + 1\right) \qquad (8)$$

  Y su varianza viene dada por:

$$var[X] = \alpha^2 \left\{ \Gamma\left(\frac{2}{\beta} + 1\right) - \left[\Gamma\left(\frac{1}{\beta} + 1\right)\right]^2 \right\} \qquad (9)$$

- **Log-Normal**: se utiliza para calcular el tiempo medio que esta un cliente en la cola.

  La función de distribución de Log-Normal viene dada por [10] :

$$p = F(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \int_0^x \frac{1}{t} exp\left\{\frac{-(log\, t - \mu^2)}{2\sigma^2}\right\} dt, para\, x > 0 \quad (10)$$

  La esperanza es:

$$e^{\left(\mu+\frac{\sigma^2}{2}\right)} \quad (11)$$

  La varianza viene dada por:

$$v = e^{(2\mu+\sigma^2)}\left(e^{\alpha^2} - 1\right) \quad (12)$$

  Donde

$$\mu = \log(m^2/\sqrt{v + m^2} \quad (13)$$

  Y su función de distribución es:

$$p = F(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \int_0^x \frac{1}{t} exp\left\{\frac{-(log\, t - \mu^2)}{2\sigma^2}\right\} dt, para\, x > 0 \quad (14)$$

2.2.3 Características de un sistema de colas, notación y terminología

Un sistema de colas se describe mediante las siguientes características [11]:

- Fuente de llegada de clientes: La llegada de los usuarios a la cola es aleatoria por lo que es necesario saber la distribución probabilística entre dos llegadas sucesivas de clientes.
- Patrón de servicio de servidores: Los servidores pueden [12] tener un tiempo de servicio variable por lo que hay que asociarlos una distribución de probabilidad ya

que puede atender en lotes o de modo individual. El tiempo de servicio varía con el número de clientes en la cola, trabajando más rápido o más lento, lo que se conoce como patrones de servicio dependientes. El patrón de servicio puede ser no-estacionario variando con el tiempo transcurrido.

- Disciplina de la cola: Es la manera en la que se van a seleccionar los clientes que se encuentran en la cola. Pueden ser:
    - *FIFO (first in first out)*: Se atiende a los clientes en orden de llegada a la cola, el primero que llega es el primero en ser atendido.
    - *LIFO (last in firs out):* el último que entra es el primero que sale.
    - *RSS (random selection of service)* o *SIRO (service in random order):* Se selecciona a los clientes de la cola de forma aleatoria.
    - *Processor Sharing:* Todos los clientes experimentan el mismo retraso, ya que soportan entre todos los clientes de la cola la capacidad del sistema atendiendo a todos por igual.
- Capacidad del sistema: Es el máximo número de clientes que puede soportar el sistema antes de ser atendidos.
- Número de canales del servicio: Para conocer este servicio se debe saber el número de servidores de éste, la distribución asociado y la distribución de probabilidad del tiempo que le lleva a cada servidor dar un servicio.
- Número de etapas de servicio: Puede tener una o varias etapas. En los sistemas con multietapa se admite vuelta atrás o reciclado por ejemplo en sistemas productivos como controles de calidad y reprocesos

Para la describir el sistema se ha usado la siguiente notación [13] : A/B/s/c/m/Z [14] donde A es la distribución del tiempo entre llegadas, B es la distribución del tiempo de servicio, s: es el número de servidores, K: es el máximo número de clientes permitidos en la cola, m: es el tamaño de la población y Z: la disciplina de la cola. A y B pueden ser:

- D: determinista o regular, es decir, cuando los clientes llegan en un intervalo de tiempo concreto.

- M: aleatorio o que sigue una distribución de Poisson.
- Ek: Erlangian, se usa en caso de que la entrada original sea una distribución de Poisson y se filtra de tal manera que solo sean admitidos lo enésimos clientes en la cola.
- G: distribuciones generales.

2.2.4 Procesos de nacimiento y muerte

Parte de los modelos de colas [15] suponen que las entradas y salidas del sistema tienen un nacimiento y muerte. El nacimiento se refiere a la llegada de un cliente al sistema de colas y la muerte se refiere a las salidas del cliente una vez que ha sido atendido.

El proceso de nacimiento y muerte se describe a través de N(t), que es el estado del sistema de la cola en el instante t, donde t = número de clientes en el sistema. En general, los nacimientos y las muertes ocurren de manera aleatoria; sus tasas medias de ocurrencia dependen del estado del sistema actual. Las hipótesis del proceso de nacimiento y muerte son las siguientes [15] [16] :

- Dada N(t)=n, la distribución del tiempo que falta para el próximo nacimiento es exponencial con parámetro λ_n donde n= (n=0,1, 2…).
- Dada N(t)=n, la distribución del tiempo que falta para la próxima salida es exponencial con parámetro μ_n donde n= (n=0,1, 2…).
- Independencia: partiendo de que las dos hipótesis anteriores son independientes, podemos definir la Transición de un estado a otro de la siguiente manera:

  $n \to n+1$ si es un nacimiento y $n \to n-1$ si es una muerte.

Tomando como supuesto que la entrada y la salida son distribuciones de Poisson los parámetros λ_n y μ_n son tasas medias.

La tasa media de llegadas al estado n implica que p (llegada de un cliente/estado n-1) * p (estado n-1) + p (salida de un cliente/estado n+1) *p (estado n+1) = la probabilidad de estar en el estado n. De lo que se deduce que la tasa media en el estado n se puede estimar por: $\lambda_{n-1} p_{n-1} + \mu_{n+1} p_{n+1}$.

Donde p_n es la probabilidad de que haya n clientes en el sistema estacionario. Al ser de esta manera la tasa media de llegadas coincide con la tasa media de salida para cualquier estado de n.

En la Figura 2.4 se muestra el estado de transiciones de nacimiento y muertes de un sistema de colas.

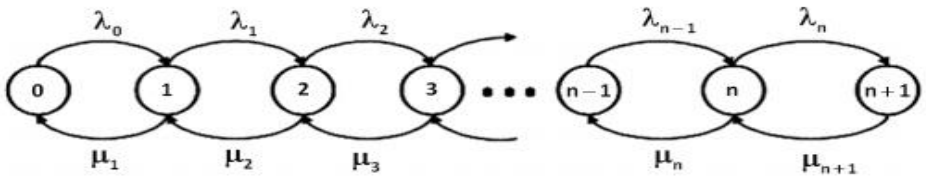


Figura 2.4 Diagrama de transiciones

Basándonos en los procesos de nacimiento y muerte [17] [18] dónde:

- Ls representa el número medio de clientes en el sistema.
- Lq se usa para saber el número medio de clientes en la cola.
- N(t) es el estado del sistema de la cola en el instante t.
- Nq(t) representa el número de clientes en la cola en el instante t.
- Ws es el tiempo de espera en el sistema para cada cliente
- Wq es el tiempo de espera en la cola para cada cliente.
- Pn(t) la probabilidad de que n clientes estén en el sistema,
- s indica el número de servidores.
- Po es la probabilidad de que no haya ningún cliente en el sistema
- λ representa la tasa de llegadas.
- μ es la tasa media de servicio del sistema.
- ρ se utiliza para calcular la ocupación del sistema donde

$$\rho = \frac{\lambda}{\mu} \qquad (15)$$

Mediante la Ley de Little [19] se relaciona el número de trabajos en el sistema con el tiempo de permanencia y su productividad o tasa de llegada, sabiendo el número medio de clientes en la cola, Lq, y en el sistema, Ls.

$$Ls = \lambda Ws \qquad (16)$$

$$Lq = \lambda Wq \qquad (17)$$

se deduce que el número medio de clientes en el sistema, Ls, es:

$$Lq + \frac{\lambda}{\mu} \qquad (19)$$

Y el tiempo medio de espera para cada cliente, Ws, es:

$$Wq + \rho \qquad (20)$$

2.2.5 Teoría de colas aplicada a nuestro trabajo

Este proyecto sigue un modelo M/Ek/1/c, es decir, se trata de un servidor con tiempos de llegadas exponenciales, una distribución de Erlang de tiempos de servicio y una capacidad c dada por el administrador de la cola. Por tanto, seguirá las siguientes fórmulas [20] :

Al ser un modelo Erlang, su función de densidad viene dado por:

$$f(t) = \frac{(\mu k)^k}{(k-1)!}\, e^{-kut} t^{k-1} \;\; para\ todo\ t \geq 0 \qquad (21)$$

Donde el tiempo medio de espera para cada cliente en el sistema es:

$$Ws = Wq + \frac{1}{\mu} \qquad (22)$$

El tiempo medio de espera en la cola para cada cliente es:

$$Wq = \lambda Ws \tag{23}$$

El número de clientes en la cola viene dado por:

$$Lq = \lambda \mathrm{Wq} \tag{24}$$

El número medio de clientes en el sistema es:

$$Ls = Lq + \frac{\lambda}{\mu} \tag{25}$$

Sabiendo que el tiempo medio de servicio es:

$$\frac{1}{\mu} \tag{26}$$

y la varianza es:

$$\frac{1}{k\mu^2} \tag{27}$$

Aplicando la fórmula de *Pollaczek-Khintchine* se deduce que

$$\mathrm{Lq} = \frac{\lambda^2\sigma^2 + \rho^2}{2(1-\rho)} = \frac{\left(\frac{\lambda^2}{k\mu^2} + \rho^2\right)}{2(1-\rho)} = \frac{1+k}{2k} = \frac{\rho^2}{1-\rho} \tag{28}$$

siendo la ocupación del sistema:

$$\rho = \frac{\lambda}{\mu} \tag{29}$$

Capítulo 3 - Herramientas

En esta sección se describen las tecnologías usadas para el desarrollo del proyecto, tanto las plataformas de desarrollo como los lenguajes de programación usados.

3.1 Herramientas de desarrollo

En este apartado se detallan las distintas herramientas de desarrollo que se han usado para poder codificar las funcionalidades de nuestra aplicación.

Para desarrollar nuestra aplicación móvil se ha utilizado la herramienta de Android Studio [21] por ser el IDE oficial de desarrollo para las aplicaciones en Android.

Las principales características de este entorno son las siguientes: perfiladores en tiempo real, que proporcionan estadísticas de la CPU, la memoria y la red de la aplicación a tiempo real; sistemas de construcción flexibles que permiten personalizar su compilación para generar múltiples variantes para diferentes dispositivos de un solo proyecto, editor de diseño visual, el cual facilita la creación de vistas de diseños complejos y el cambio de las configuraciones según el dispositivo móvil en el que estemos, emulador rápido, que simula diferentes características y configuraciones de Google para crear experiencias de realidad aumentada, y su analizador de apks, para reducir el tamaño de la aplicación de Android inspeccionando el contenido de su archivo apk.

Para implementar las funcionalidades se ha empleado Kotlin, que es un lenguaje de programación de tipo estático, creado por la compañía JetBrains y el lenguaje oficial de Android desde 2017.

Se eligió Kotlin [22] para desarrollar el proyecto frente a Java por las ventajas que ofrece. Por una parte, permite escribir un código legible y sencillo, es más seguro, ya que se pueden controlar las excepciones de tipo "*NullPointerException*", y cuenta con mecanismos para evitar errores comunes en tiempo de ejecución y compilación. Además, es compatible con versiones antiguas en dispositivos Android a partir de la versión JDK 6. Es un lenguaje interoperable con Java y compatible para dispositivos basados en SO iOS.

3.2 Herramientas de servicio

Las distintas herramientas usadas para desarrollar las partes relacionadas con el servicio son las que se detallan a continuación.

En primer lugar, se ha creado nuestro servicio usando Spring. Esta herramienta es un *framework* de código abierto [23] para construir aplicaciones empresariales en Java de alto rendimiento, es liviano y reutilizable debido a que su finalidad es estandarizar, facilitar y resolver los problemas que vayan surgiendo durante el desarrollo.

Spring [24] y Spring Boot se basan en la filosofía de convención sobre configuración, es decir, reducir al mínimo el número de pasos que debe dar el desarrollador en la configuración inicial.

En nuestro proyecto se ha usado Spring Boot [25] que es una solución a Spring y facilita la creación de aplicaciones independientes basadas en Spring.

Algunas de sus características son: la incorporación de aplicaciones de servidores web y contenedores, simplificación de la configuración Maven y configuración automática de Spring.

Además, se ha utilizado IntelIj para desarrollar la parte del servidor en Java, creado por JetBrains, ya que algunas de sus características son bastante útiles para nuestra aplicación como la compatibilidad con los lenguajes de programación Java, Kotlin, Groovy y Scala, las aplicaciones Android, las herramientas para bases de datos SQL con la que hemos configurado la nuestra a través de PostgreSQL y la integración de las tecnologías de Spring y Spring Boot, además de las especificaciones Swagger.

También de los desarrolladores de IntelIj se ha utilizado DataGrip, una herramienta multiplataforma dirigida a los desarrolladores de bases de datos SQL. Además, contiene un Sistema de Gestión de Bases de Datos o SGDB [26], con un conector de bases de datos de Java permitiendo conectar con el proyecto en Java. La utilidad de DataGrip en nuestro proyecto ha sido en la creación y visualización de nuestra base de datos, y en la elaboración de las consultas necesarias para sacar la información sobre la aplicación.

Conectados la base de datos y el proyecto, se ha utilizado Postman [27] para hacer peticiones HTTP como *GET, POST, PUT, DELETE* Y *UPDATE* a una dirección de nuestro interés

ya que esto es útil para testear nuestros desarrollos e interactuar con APIs. Las principales características de esta herramienta son: la accesibilidad, permitir al usuario el uso de colecciones para las llamadas a la API; la verificación de pruebas, para saber si la solicitud HTTP que se ha realizado ha devuelto lo que debe; y la depuración para saber qué datos han sido recuperados. Principalmente ha sido usada para testear las peticiones al servidor tanto en local como en remoto.

En cuanto a la base de datos del proyecto, en primera instancia se pensó utilizar los servicios de Google Firebase, pero no supuso una solución dada la falta de entidades relacionales en las bases de datos que ofrecía. Por eso decidimos cambiar a PostgreSQL, gracias a nuestros conocimientos de SQL y la fácil conexión que tiene con nuestro proyecto y Heroku.

PostgreSQL [28] es un sistema de bases de datos relacionales de código abierto y gratuito que ofrece varías características en cuanto a los tipos de datos permitiendo: definir datos primitivos, estructurados y documentos. En cuanto a la concurrencia y rendimiento, algunas de sus características son: control de concurrencia de versiones múltiples, indexación, planificación y optimización de consultas y transacciones. Por último, posibilita funciones y procedimientos almacenados y lenguajes de procedimiento como: PL / PgSQL, Perl, Python, expresiones de ruta SQL/Json y contenedores de datos externos que facilita conéctese a otras bases de datos o secuencias con una interfaz SQL estándar.

Aparte de conectar PostgreSQL con nuestra aplicación, también se ha conectado a Heroku. Éste es una plataforma como servicio de computación en la Nube (PaaS) [29] que sirve para desplegar y ejecutar aplicaciones.

Heroku ejecuta sus aplicaciones dentro de contenedores inteligentes llamados *dynos* dónde se pude desplegar código en los siguientes lenguajes Java, Node, PHP o Python. También dispone de complementos [30] llamados *add-on*, que son servicios en la nube que añaden funcionalidades extra a las aplicaciones. Para implementar la base de datos de PostgreSQL dentro de Heroku se ha usado el *add-on* Heroku PostgresSQL.

3.3 Herramientas de gestión

Trello es una herramienta de gestión de proyectos que hace que la colaboración sea sencilla. Puede usarse en cualquier ámbito de la vida, desde un proyecto de gran calibre hasta un viaje entre amigos.

Se decidió usar esta herramienta para organizar los distintos *sprints* que se han realizado a lo largo del año. El tablero se ha llamado QFLOW y la finalidad de cada columna se explica a continuación.

- *Product Backlog*: son las tareas que se van añadiendo en función de su necesidad a lo largo del proyecto. Estas tareas se han ido añadiendo a la columna siguiente en función de si las habían considerado prioritarias para el *sprint* o no.
- Tareas a realizar: son los objetivos que se han añadido para completar las tareas durante el *sprint*.
- En proceso: hace referencia a las tareas que están siendo realizadas entre los distintos *Sprint Reviews*. Este es el paso anterior para tener cada tarea completada.
- Completadas durante el *sprint*: se añaden las tareas que han sido completadas, al *sprint X* correspondiente.
- *Sprint* X: es la última columna donde se añaden los objetivos completados durante ese *sprint*. Siendo X el número de *sprint* correspondiente.

Para organizar las llamadas en grupo y conectar los distintos ficheros se ha usado Microsoft Teams. Esta herramienta es una plataforma creada por Microsoft que sustenta el trabajo en equipo en las empresas siendo un tipo de software que pone a disposición salas de chat, fuentes de noticias y grupos. Forma parte del paquete 365 de Microsoft Office y no puede ser configurado individualmente. Con él se pueden hacer vídeos por Twitch, compartir archivos, y acceder al Bloc de notas, IPages, Powerpoint y OneNote.

Cabe destacar que al principio hubo problemas a la hora de escribir la memoria por la colisión de distintas versiones, faltaban partes que ya habían sido escritas previamente y había que prestar demasiada atención a si un compañero había modificado previamente el documento.

Por ello, Microsoft Teams ha sido la herramienta utilizada para compartir la memoria, lo que nos ha permitido tener una buena sincronía a la hora de ir actualizando los distintos apartados de la ésta.

Como conclusión, se puede decir que esta herramienta ha sido útil para la realización de nuestro trabajo de fin de grado.

Para la unión de todo nuestro código se ha usado GitHub. Esta herramienta es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git, que ha sido utilizada en numerosas ocasiones para organizar el código de las distintas prácticas que han sido realizadas durante la carrera. Gracias a esto ya teníamos las nociones básicas para su uso.

Los proyectos se han organizado en repositorios, que pueden ser públicos o privados. Un repositorio es un directorio donde están alojado los distintos ficheros relacionados con la aplicación**.**

Por último, queríamos tener una documentación de APIs estable y correcta, por lo que se decidió usar Swagger. Esta herramienta es un *framework* para documentar APIs Rest desde diferentes fuentes: archivos de configuración, XML, C#, Javascript, Ruby, PHP, Java, Scala y otros. Además, existen multitud de módulos que pueden ayudar a integrarlo en tu proyecto. Gracias a la facilidad de uso de esta herramienta, documentar nuestra API ha sido sencillo.

Por último, pero no menos importante, se ha usado Git Kraken, que es una herramienta utilizada en empresas como Netflix, Tesla y Apple. Git Kraken es una interfaz gráfica multiplataforma para Git, con la que podemos tener un mapa visual de nuestras ramas junto a sus *commits*, lo que nos permite controlar de una manera más eficaz las distintas versiones que se van subiendo. Es una interfaz muy sencilla que utiliza la técnica de *drag and drop* para hacer los *merge* entre ramas.

En nuestro caso se ha usado esta herramienta para seguir más concienzudamente el avance del proyecto. La facilidad a la hora de realizar *commits* y el poder solucionar la colisión de versiones ha sido lo que más nos ha llamado la atención. Gracias a que algunos miembros del grupo ya habían utilizado esta herramienta, su integración en el equipo fue bastante sencilla.

GitFlow es la metodología con la que se ha podido llevar el *workflow* de una forma más consistente y firme, permitiendo así que las ramas más importantes, como *dev* o *master*, no queden en segundo plano y no se pierdan versiones.

Capítulo 4 - Casos de Uso

En este capítulo se detallan los casos de uso de la aplicación mediante los que se describen los servicios y las restricciones de un sistema, es decir, los casos de uso describen la forma en la que los tipos de usuarios o sistema externo interactúan con el sistema [31].

Como se comentaba en la introducción, se ha conseguido elaborar el producto viable mínimo [32] de nuestro proyecto, ha consistido en crear un ecosistema en el que mediante el uso de aplicaciones móviles un usuario sin privilegios puede unirse a una cola creada por un usuario con rol de administrador, el cual tiene las capacidades de crear y gestionar una cola.

Como en todas las aplicaciones de hoy en día se necesita un caso de uso para la creación de usuarios y administradores como se muestra en la siguiente Tabla 4.1.

Tabla 4.1 Caso de uso del registro de un usuario

Registro	
Objetivo	Dar de alta un usuario.
Entrada	Obligatorios: nombre, apellido, email y contraseña.
Precondición	El email no esté dado de alta.
Salidas	Mensaje confirmando el registro.
Postcondición (Éxito)	El sistema da de alta al usuario en la base de datos.
Postcondición (Fallo)	El sistema manda un mensaje informando del error.
Actores	Usuario.
Descripción	El sistema muestra la vista del registro con los campos que se deben rellenar. El usuario completa los campos obligatorios: nombre, apellido, email y contraseña. El sistema comprueba que los datos introducidos son correctos y da de alta al usuario, en caso contrario el sistema informa al usuario a través de un mensaje.

A continuación, se muestra la descripción del inicio de sesión válido para los dos tipos de usuario que usa nuestra aplicación (Tabla 4.2).

Tabla 4.2 Caso de uso Login

Login	
Objetivo	Iniciar sesión.
Entrada	Email y contraseña.
Precondición	Estar registrado.
Salidas	Mensaje de inicio de sesión correctamente.
Postcondición (Éxito)	Guardar la sesión.
Postcondición (Fallo)	Avisar al usuario que el correo o la contraseña introducidos son incorrectos.
Actores	Usuario.
Descripción	El usuario rellena los campos de email y contraseña. Si los datos son correctos accede a la vista principal, si no muestra un mensaje de error solicitando al usuario que vuelva a rellenar los datos correctamente.

En este caso de uso se describe como un administrador crea una cola rellenando los campos correspondientes para poder darla de alta (Tabla 4.3).

Tabla 4.3 Caso de uso dar de alta una cola.

Alta Cola	
Objetivo	Dar de alta una cola.
Entrada	Obligatorios: nombre, descripción, capacidad, negocio asociado, si está bloqueada.
Precondición	Administrador registrado y logueado.
Salidas	Mensaje con confirmación.
Postcondición (Éxito)	Se crea una cola nueva.
Postcondición (Fallo)	Se muestra un mensaje de error.
Actores	Administrador y cola.
Descripción	El administrador rellena los campos para crear una cola nueva, en la pantalla principal se mostrarán algunos de los datos de la cola creada. Si no hay éxito saldrá un mensaje de error.

Mediante la Tabla 4.4 se describe como es el proceso de dar de baja una cola por el administrador que la ha creado.

Tabla 4.4 Caso de Uso dar de baja una cola.

Baja de cola	
Objetivo	Eliminar cola que está dada de alta.
Entrada	Fecha de cuando acaba el evento y el estado de la cola bloqueada pasa a ser cierto.
Precondición	Administrador logueado.
Salidas	Mensaje de confirmación.
Postcondición (Éxito)	Se elimina la cola seleccionada. Se muestra un mensaje confirmando la eliminación.
Postcondición (Fallo)	Mensaje de error de salida.
Actores	Administrador y cola.
Descripción	El administrador selecciona la cola que quiere quitar, si hay éxito muestra un mensaje confirmando la eliminación de la cola. Si no se muestra un mensaje de error.

La Tabla 4.5 explica lo que ocurre cuando un usuario registrado y logueado se une a una cola.

Tabla 4.5 Caso de uso para unirse a una cola.

Unirse Cola	
Objetivo	Añadir a un usuario a la cola.
Entrada	Datos del usuario logueado.
Precondición	Usuario logueado.
Salidas	Mensaje de confirmación.
Postcondición (Éxito)	Se añade al usuario a la cola. Se muestra en su pantalla la cola a la que se ha unido.
Postcondición (Fallo)	Se muestra un mensaje de error.
Actores	Usuario y cola.
Descripción	El usuario después de añadir el código de una cola pulsa el botón de unirse. Si hay éxito se le añade, mostrándole los datos de la cola. Si no lo hay se muestra un mensaje de error.

En la Tabla 4.6 se describe la información relacionada con las colas para los usuarios.

Tabla 4.6 Caso de uso ver información de la cola

Ver Información colas (Usuario)	
Objetivo	Muestra la información de una cola.
Entrada	Datos del usuario logueado e información de la cola.
Precondición	Estar logueado previamente.
Salidas	Ninguna.
Postcondición (Éxito)	Muestra correctamente los datos.
Postcondición (Fallo)	Muestra un mensaje de error.
Actores	Usuario.
Descripción	Se muestra si los usuarios se quieren unir a la cola, si es su turno y el tiempo medio de espera en la cola.

En la Tabla 4.7 se muestra la información relacionada con las colas para los administradores.

Tabla 4.7 Caso de uso ver información cola administradores

Ver Información colas (Administrador)	
Objetivo	Muestra la información de una cola.
Entrada	Datos del administrador logueado e información de la cola.
Precondición	Estar logueado previamente.
Salidas	Ninguna.
Postcondición (Éxito)	Muestra correctamente los datos.
Postcondición (Fallo)	Muestra un mensaje de error.
Actores	Administrador.
Descripción	Muestra el QR, el código de la cola, las personas que hay en la cola.

En la Tabla 4.8 se detallan algunas de las acciones que puede realizar el administrador para gestionar una cola.

Tabla 4.8 Caso de uso para la gestión de una cola

Gestión de la cola (Administrador)	
Objetivo	Permite a un administrador acceder a todas las funcionalidades de una cola en proceso.
Entrada	Datos del usuario logueado e información de la cola.
Precondición	Estar logueado previamente y tener una cola activa.
Salidas	Ninguna.
Postcondición (Éxito)	Muestra correctamente las opciones.
Postcondición (Fallo)	Muestra un mensaje de error.
Actores	Administrador y cola.
Descripción	Permite al administrador realizar las siguientes acciones con la cola: bloquearla, eliminar a alguien de la cola, pasar a la siguiente persona... y ver el número de personas en la cola.

Un administrador avanza la cola cuando ya ha pasado el turno de un usuario o cuando se salta el usuario ya sea porque este no esté en la cola o se le haya expulsado de está por alguna razón como se muestra en la Tabla 4.9.

Tabla 4.9 caso de uso avanzar cola

Avanzar cola	
Objetivo	Pasar al siguiente usuario en la cola.
Entrada	Cola, usuarios en la cola.
Precondición	Administrador logueado y gestor de esa cola.
Salidas	Cola actualizada.
Postcondición (Éxito)	La cola se actualiza.
Postcondición (Fallo)	Mensaje de fallo.
Actores	Administrador y cola
Descripción	Un administrador responsable de la cola puede hacer que se avance la cola.

Una de las gestiones que puede realizar el administrador de una cola desde su panel de control es la posibilidad de reanudar una cola como se muestra en la Tabla 4.10.

Tabla 4.10 Caso de uso de reanudar una cola

Reanudar cola	
Objetivo	Reactivación de una cola.
Entrada	Cambiar el estado de una cola.
Precondición	Ser administrador.
Salidas	Mensaje de confirmación.
Postcondición (Éxito)	Reactivación de una cola.
Postcondición (Fallo)	Se mandará un mensaje de error.
Actores	Administrador y cola.
Descripción	El administrador de la cola vuelve activar la cola.

A continuación, en la Tabla 4.11 se describe cuando un administrador bloquea la cola por alguna razón, como el instante en el que se hace un descanso, donde la cola se volverá abrir cuando este haya terminado y se avisará a los usuarios.

Tabla 4.11 Caso de uso bloquear cola

Bloquear cola	
Objetivo	Cerrar el acceso a la cola.
Entrada	Cola y administrador.
Precondición	La cola debe estar activa y ser administrador.
Salidas	Mensaje de éxito.
Postcondición (Éxito)	La cola bloqueada.
Postcondición (Fallo)	Mensaje de error.
Actores	Administrador y cola.
Descripción	Se bloquea la cola hasta próximo aviso manteniendo su estado actual y se impide que nuevos usuarios se unan.

En la Tabla 4.12 se explica lo que un usuario o un administrador ve desde su perfil de *Home.*

Tabla 4.12 Caso de uso ver perfil.

Ver Home	
Objetivo	Mostrar la pantalla principal.
Entrada	El usuario y datos sobre sus colas de espera.
Precondición	Estar logueado.
Salidas	Ninguna.
Postcondición (Éxito)	Ninguna.
Postcondición (Fallo)	Volver a iniciar sesión.
Actores	Usuario y colas.
Descripción	Muestra la vista principal que recibe un usuario nada más iniciar sesión indicando si se encuentra en una cola, además de un listado de las colas en las que ha estado.

Capítulo 5 - Servicios web y bases de datos

En este capítulo se ha detallado cómo ha sido desarrollada la base de datos de la aplicación, así como los servicios web que se han utilizado para conseguir que la base de datos sea común a todos los usuarios y esta se actualice a tiempo real.

El diseño de la base de datos se ha realizado para tener a fácil disposición de cualquiera de los datos necesarios para la realización de tareas de forma sencilla, como se muestra en la Figura 5.1.

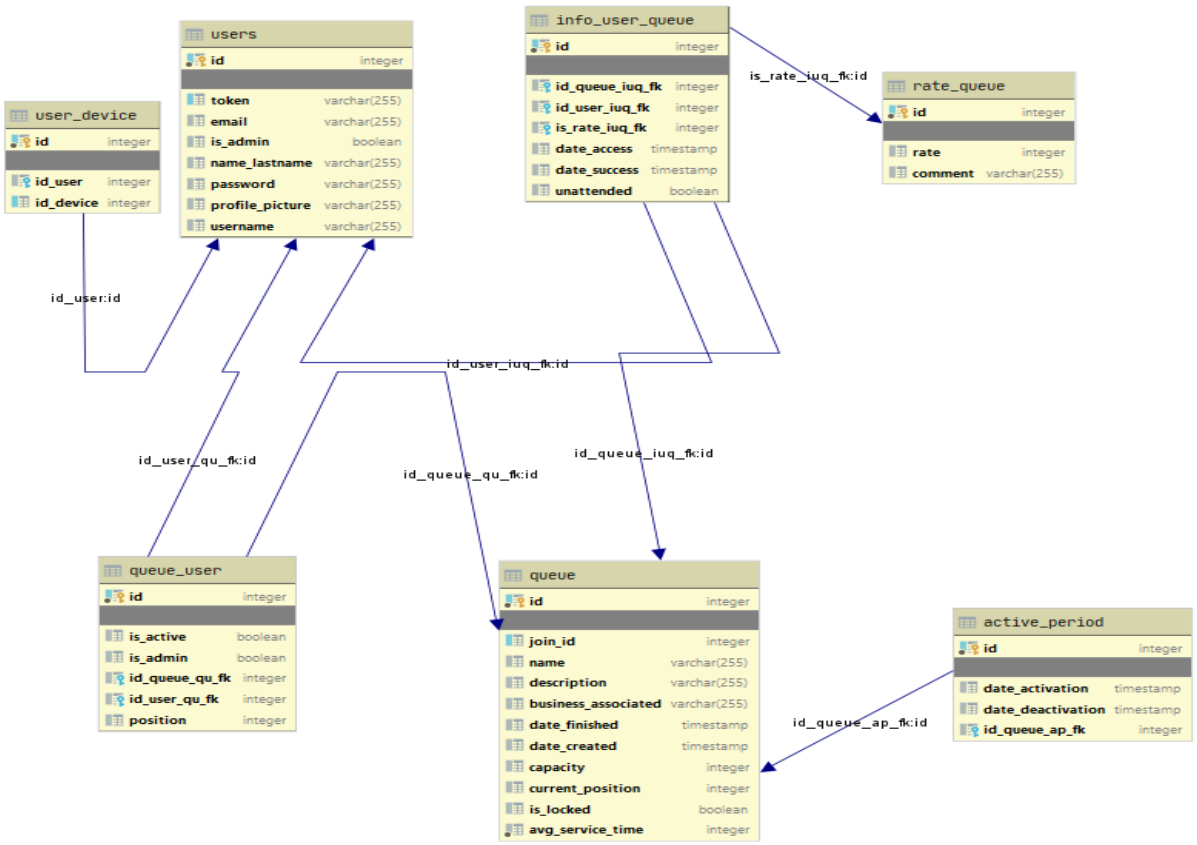


Figura 5.1 BBDD PostgreSQL desde DataGrip

Este diseño ha sido esencial para que el desarrollo de la aplicación haya sido fluido. A continuación, se comenta tabla a tabla como se ha diseñado la base de datos para el desarrollo del proyecto.

En la **tabla *users*** se guardan los perfiles tanto de los usuarios sin privilegio que quieran acceder a colas como los de los administradores que puedan crear colas o modificar las ya existentes.

Los valores que se han considerado necesarios a la hora de crear un perfil son: un *id* generado automáticamente y de forma aleatoria por PostgreSQL, que ha sido utilizado para reconocer al usuario y hacer *queries* sobre la base de datos; el *token* de cada usuario, para relacionar un usuario en la aplicación; el *username*, nombre del usuario que ha sido elegido por éste en el formulario de registro; el *name_lastname*, nombre y apellido real del usuario; el correo electrónico del usuario, que se guarda en *email* y su contraseña en *password; profile_picture*, que guarda la aplicación y por último se guarda *is_admin*, que es un valor booleano que determina si el usuario tiene el poder de crear colas.

En la **tabla *queue*** se han almacenado todos los aspectos importantes a tener en cuenta a la hora de la creación, mantenimiento y cierre de una cola por parte de un administrador.

Se almacenan los siguientes campos: *id*, generado automáticamente de forma aleatoria por PostgreSQL, usado para reconocer la cola a tratar y hacer *queries* sobre la base de datos; *join_id*, índice generado para poder unirse a una cola; *name*, nombre de la cola; *description*, descripción del puesto al que se espera en su cola; *capacity* es la capacidad total de la cola, determinado por el administrador de esta; *business_associated*, empresa a la que está asociada el puesto al que se espera; *date_created* y *date_finished* , donde se guardan las fechas de creación y finalización de la cola; *current_position*, que representa la posición actual de la cola; *is_locked*, valor booleano que indica si la cola está bloqueada o no; y finalmente, *avg_service_time* que define el tiempo de espera promedio estimado por el creador de la cola. Se da la posibilidad de bloquear la cola al administrador para poder, por ejemplo, pausarla en caso de que se quieran hacer recesos en la actividad del puesto.

En la **tabla *active_period*** se han guardado los periodos de activación que ha tenido una cola para poder implementar el uso de periodos de receso de la actividad en las colas. Para llenar ese cometido se almacenan los siguientes valores por tiempo de activación de la cola.

Los parámetros de la tabla son: *id*, identificador creado de forma automática por PostgreSQL; *id_queue_ap_fk*, clave foránea que nos indica la cola que ha sido activada en este

periodo; *date_activation*, fecha de activación de la cola y por último *date_desactivation*, fecha de desactivación de la cola.

En la **tabla *queue_user*** se ha usado esta colección para guardar toda la información relacionada entre una cola y un usuario, es decir, para todo aquel usuario que, en algún momento, sea presente o pasado, haya estado esperando virtualmente en esa cola.

Los parámetros que se guardan en la tabla son: *id*, generado por PostgreSQL de forma aleatoria; *id_user_qu_fk*, clave foránea referente al usuario que está registrado en esa cola; *id_queue_qu_fk*, identificador de la cola sobre la que se refiere la tabla; *is_admin*, valor booleano que determina si el usuario de la relación es administrador de la cola; *is_active*, otro valor *booleano* que informa si el usuario está en ese momento esperando dentro de la cola o si por el contrario ya ha salido de la cola y la relación es antigua; y por último *position*, para saber cuál es la posición de un usuario en la cola.

En la **tabla *info_user_queue*** se han almacenado más datos importantes que necesitaremos a la hora de realizar en una futura iteración diferentes estadísticas sobre el comportamiento de los usuarios dentro de las colas.

Almacenamos por tanto los siguientes valores: *id*, generado automáticamente por PostgreSQL de forma aleatoria; *id_queue_iuq_fk*, clave foránea de la colección; *id_user_iuq_fk*, para saber a qué relación de usuario y cola nos estamos refiriendo*; is_rate_iuq_fk*, clave foránea que indica el identificador de la calificación que dio el usuario al puesto, explicada en la siguiente colección. El parámetro *date_access*, que representa la fecha en la que el usuario accedió a la cola. De forma parecida, *date_success* guarda cuando el usuario ha completado la cola y ha visitado el puesto, fecha en la que esto ocurrió; por último, *unnatended* se utiliza para saber si un usuario ha cancelado el turno.

La **tabla *rate_queue*** se utilizará para poder guardar información adicional, en este caso un comentario, respecto a la calificación que un usuario de a un puesto que ha visitado después de realizar la cola virtual y pasar por el puesto. Los valores que se guardan en la tabla son: *id*, que genera PostgreSQL de forma automática para identificar a la calificación; *rate* representa la calificación numérica que el usuario deja respecto a su experiencia del puesto que ha visitado; por último, *comment*, que es el comentario que ha realizado el usuario.

Por último, la **tabla *user_device*** se podrá usar para guardar los dispositivos desde los que se ha iniciado sesión, con los parámetros: *id*, que genera PostgreSQL de forma automática; *id_user*, clave foránea referente al usuario que ha iniciado sesión con ese dispositivo e *id_device*, identificador asociado al dispositivo que inicia sesión.

Capítulo 6 - Aplicación Móvil

En este capítulo se ha presentado todo lo relacionado con el desarrollo de la aplicación móvil de nuestro proyecto; esto comprende la definición de la aplicación para que cumpla una funcionalidad mínima por la que un usuario pueda unirse a una cola y que un administrador pueda gestionarla, es decir, crear, parar, cerrar, reanudarla y avanzar una cola.

La organización de este apartado de la memoria ha consistido en definir la estructura de la aplicación junto con las tecnologías que se han utilizado, continuando por el diseño planteado y finalizando con el producto adquirido y su uso.

6.1 Arquitectura de la aplicación

En el desarrollo de aplicaciones Android, existe una problemática causada por la persistencia de objetos, debido a los ciclos de vida por los que pasa una aplicación. Esto lleva a los desarrolladores a aplicar ciertas arquitecturas relativamente complejas con el fin de asegurar un funcionamiento correcto.

Además, para garantizar la calidad del software, se debe asegurar que durante el desarrollo se siguen los principios *SOLID* [33], los cuales, de manera resumida, son los principios básicos de la programación orientada a objetos.

Por esto, en este apartado, se van a comentar los patrones que se han utilizado junto con la motivación de su uso. Se continuará mostrando un ejemplo de la ruta a seguir a partir de estos patrones y finalmente se explicará la separación en dos aplicaciones diferentes mediante la utilización de *Flavours*.

6.1.1 Patrones utilizados

Con la problemática del ciclo de vida en mente, se planteó la utilización de tres patrones de diseño, el *Model View ViewModel* [34], imprescindible para la persistencia de estos objetos, el patrón de *Clean Architecture* [35], se encarga de modularizar correctamente el código de la aplicación, y la inyección de dependencias, con Koin [36], para permitir el uso de ciertos objetos entre las diferentes capas de nuestra aplicación

El patrón *Model View ViewModel* se encarga de asegurar la persistencia de los objetos creados en tiempo de ejecución de la aplicación. Una explicación del problema que conlleva a describir cómo funciona el ciclo de vida [37] de una aplicación de Android.

Como se puede apreciar en la Figura 6.1, una aplicación pasa por diferentes etapas desde su creación hasta su destrucción, pasando por paradas intermedias en ciertas circunstancias, como el salir de la aplicación, el giro de pantalla, o el bloqueo del dispositivo. Además, para completar esta explicación es importante mencionar que en el proceso de cambio de los estados se produce con la llamada a diferentes funciones.

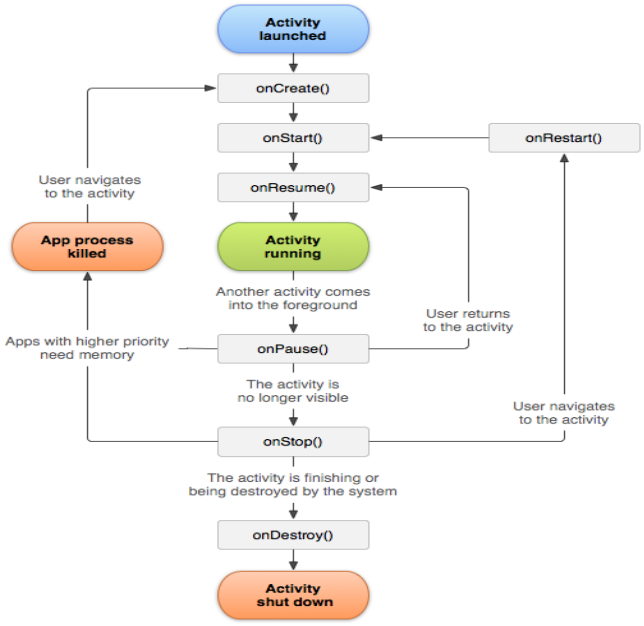


Figura 6.1 Ciclo de vida de una actividad Android

Comprendido esto, el problema surge en el momento en que se produce una transición desde la parada de una pantalla a su reanudación, ya que se puede destruir la vista que en ese momento se encontraba activa, con todos los objetos que contenía. Esto lleva a la creación de una entidad, conocida como *ViewModel*, que se encarga de realizar toda la lógica que respalda a la vista.

El patrón que más se utiliza y que implementa a esta es *Model View ViewModel*. Consiste en comunicar la vista con el *ViewModel* de manera unidireccional, por lo que la primera puede llamar y acceder al *ViewModel*, sin dar la posibilidad de que ocurra lo contrario. Una buena manera de visualizar esto lo podemos encontrar en la Figura 6.2.

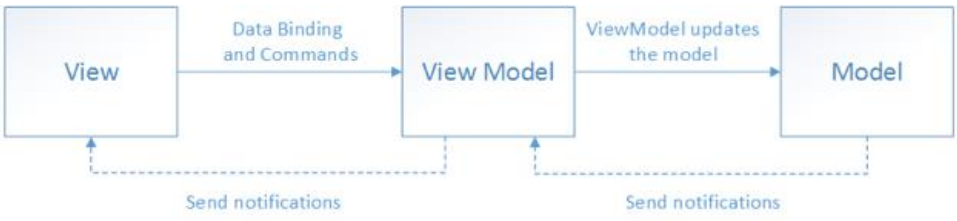


Figura 6.2 Comportamiento valor ViewModel

En el caso en el que se quiera transmitir algún dato a la vista, esta deberá actualizarse a través de los observadores que serán activados mediante cambios en la *ViewModel*. Esta conexión se replica exactamente igual para el *ViewModel* y el modelo de la aplicación, de manera que queda bien separada la lógica de la aplicación y las vistas, asegurándonos así de que en el caso de perder la vista, no se genere un estado de error en la aplicación

Con la aplicación de este patrón, se solventa el problema creado por los ciclos de vida de una aplicación de Android, separando correctamente la vista de la lógica de la aplicación, lo que conlleva que mucho código se condense en la sección del modelo y de los *ViewModel* ya que entre ambos se realiza todo lo relacionado con la funcionalidad de la aplicación.

Esto genera la necesidad de modularizar todo lo realizado por estos dos elementos en partes más diferenciadas, que se encarguen de aspectos más específicos de la lógica de la aplicación. Para ello se ha decidido utilizar el patrón *Clean Architecture*, porque es uno de las más utilizados en las aplicaciones móviles.

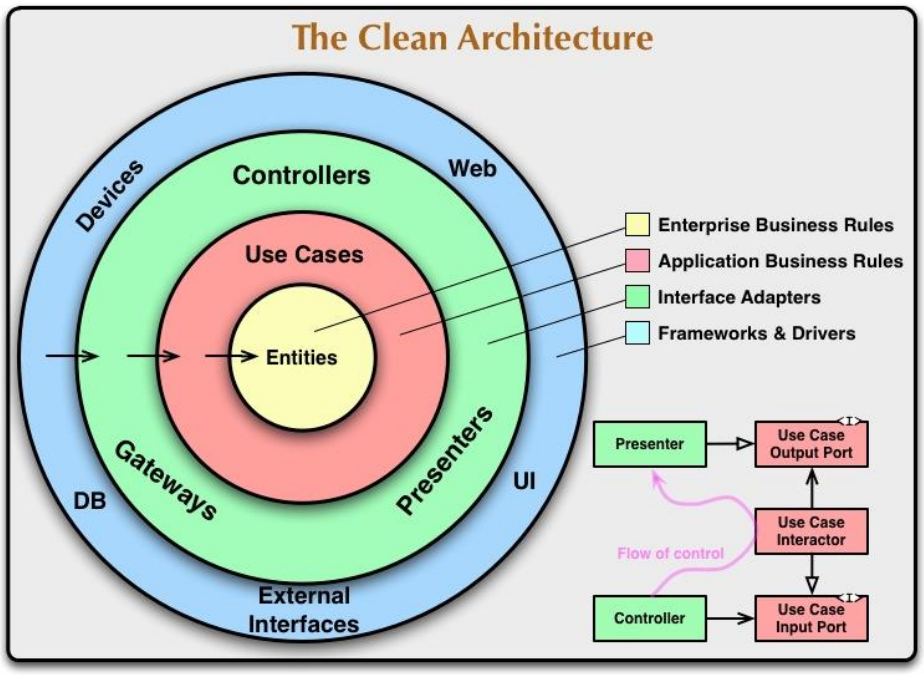


Figura 6.3 Diagrama Arquitectura Clean

El diagrama de la Figura 6.3 representa cómo se organiza la estructura de este patrón, el cual podría considerarse una extensión del *Model View ViewModel*, dada su unilateralidad y organización.

Lo que añade la Arquitectura *Clean* es la separación de la lógica del *ViewModel* en casos de uso que, a su vez, conectan con los repositorios donde se encuentra la información del modelo, necesaria para llevar a cabo una funcionalidad.

Para evitar cualquier posible confusión llamaremos *Use Cases* a las entidades relacionadas con la Arquitectura *Clean*, manteniendo los casos de uso para las definiciones de la aplicación referidas anteriormente.

Como se puede observar, gracias a los dos patrones anteriores, se ha solventado tanto el problema de la persistencia de los objetos como la separación de las funcionalidades acorde a los principios *SOLID*.

Sin embargo, con el objetivo de conseguir esto, se ha creado la necesidad de entregar objetos a través de las capas. Por ejemplo, los *Use Cases* los necesitaremos en los *ViewModels* y, dado que estos harán conexiones con los repositorios, deben de ser capaces de obtenerlos. Esto puede ser problemático de cómo los van a obtener si los instanciamos en el *ViewModel*, donde un repositorio nunca debe de ser accesible.

Aquí es donde entra en juego el inyector de dependencias, que se encarga de generar una sola instancia de cada clase que necesitemos, y le entrega los parámetros que necesite para su funcionamiento. Así desde el *ViewModel* obtenemos los *Use Cases* que necesite, y estos tendrán a su vez aquellos objetos que necesiten.

De esta manera si necesitamos un *Use Case* que utilice un repositorio, con la obtención del *Use Case*, tendremos la funcionalidad al completo sin habernos saltado ninguna de las capas de nuestra arquitectura.

6.1.2 Ejemplo de la utilización de los patrones

Para ejemplificar los patrones que se han utilizado en la aplicación, vamos a explicar el flujo que seguiría la implementación de una funcionalidad sencilla, como puede ser el caso de uso del *Login*.

En este ejemplo aparecerán, la vista, que está relacionado con su Actividad, que tiene asociada un *ViewModel*, este a su vez habrá recibido por inyección de dependencias el *Use Case* del *Login*, que contiene el repositorio de los usuarios, para realizar todo lo relacionado con el servidor.

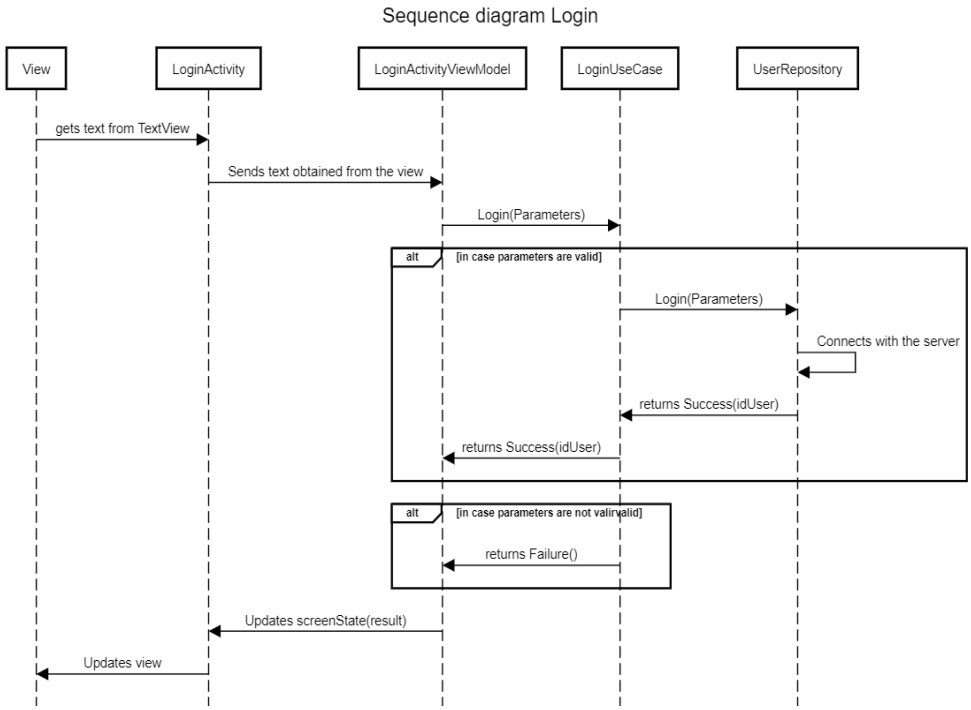


Figura 6.4 Diagrama secuencia Login

La Figura 6.4 indica cómo sería el flujo. Originalmente los datos para el *Login* se van a encontrar en la vista, los cuales se obtendrán de la actividad *LoginActivity*. Ésta llamará al *ViewModel* para que realice la función del *Login*, enviando los datos obtenidos previamente.

El *ViewModel*, liberado de carga de trabajo, simplemente llamará al *Use Case* del *Login*, que ya implementará toda la lógica, permitiendo que la cantidad de código que hay en esta clase sea más reducido.

El *Use Case* hará todas las comprobaciones que están en la aplicación, como que el email sea realmente un email, que la contraseña no sea demasiado corta, o que no haya ningún carácter no permitido.

Una vez se pasa esta validación, si es que se supera, este llamará al repositorio que se encargará de llamar al servidor, gracias a la clase *ApiService*, de la que hablaremos más adelante.

A continuación, el servidor responde, y el repositorio con ayuda de nuestra función *request* en el *BaseRepository*, recibe la respuesta como un *String* de manera que, tras convertirlo en un objeto propio de la aplicación, va hacia el caso de uso que a su vez llama a la función de éxito del *ViewModel* correspondiente.

Esta función actualizará el valor del objeto observado por la vista, que al ver que cambia el valor, iniciará el funcionamiento acorde a un *Login* correcto. Es decir, la navegación a la vista del Home.

6.1.3 *Flavours* de la aplicación

Con los patrones explicados anteriormente hemos asegurado una arquitectura que solventa una de las mayores dificultades del desarrollo Android y hemos asegurado una buena diferenciación entre las partes de la aplicación, asegurando que esta diferenciación sea consistente en todo el código.

No obstante, esta modularización permite observar que la complejidad de la aplicación, tal y como la teníamos planteada es relativamente alta. De hecho, al poco tiempo de empezar el desarrollo, fácilmente se llegaba a más de 40 clases donde se acumulaba mucha carga de trabajo en cada una de las vistas junto a sus *ViewModels*.

Esto se debía principalmente a la funcionalidad diferente que tenía que ser implementada para cada uno de los roles de usuario. Por esto que se empezó a valorar la posibilidad de desarrollar dos aplicaciones paralelas, ya que, al tener unas funcionalidades tan diferenciadas, se empezó a dejar de pensar que una aplicación común fuese la opción más correcta.

Con esta idea en mente, se estudió su viabilidad de cara al desarrollo. Lo cual mostró atisbos de ser posible gracias a la capacidad de Android Studio de crear *Build Variants* [38].

Los *Build Variants*, permiten añadir *Flavours* a la aplicación, siendo estas pequeñas variaciones de productos que permiten tener una parte común y tener otra completamente específica.

Conociendo esta posibilidad se ha tomado la decisión de aplicar esta opción y desarrollar dos aplicaciones. Una que implemente el rol de los Administradores, y otra que implemente aquello que un usuario normal necesite.

Una explicación más exhaustiva de las clases que se han utilizado en el proyecto se puede encontrar en el Anexo 2.

6.2 Estructura final del proyecto

Después de elegirlos patrones que han sido utilizados, y tras haber tomado la decisión de separar la aplicación en dos, además de haber explicado todas las clases principales que han sido utilizadas durante el desarrollo, la arquitectura final ha quedado como se muestra en la Figura 6.5.

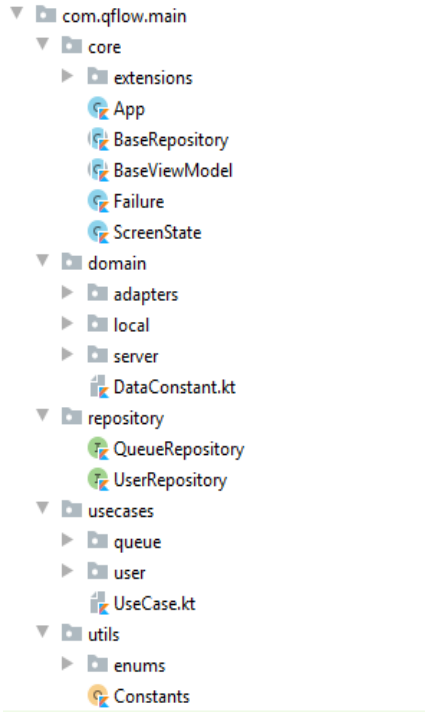


Figura 6.5 Estructura paquete Main

El paquete *Main*, es común para nuestras dos aplicaciones, y contiene:

- *Core*: donde se encuentran las clases importantes del núcleo de nuestra aplicación, como la clase *App.kt* que supone la inicialización de todo aquello que persista mientras la aplicación esté en curso.
- *DInjector*: el cual contiene la clase *Koin*, que configura el inyector de dependencias del proyecto. Es probable que más adelante éste se tenga que separar por *Flavour*
- *Domain*: el lugar al que pertenecen todas las clases de dominio, tanto local, como en nuestro servidor. Es decir, la representación en código de las clases que intervienen en ambas aplicaciones. Además, encontramos los *adapters*, que nos permiten pasar estas de una a otra.
- *Repository*: Aquí se encuentran las clases que conectan con las bases de datos y servidores.
- *Use Cases*: como su nombre indica, guardaran los casos de uso.
- *Utils*: carpeta donde va todo lo que no pertenece a ninguna de las anteriores.

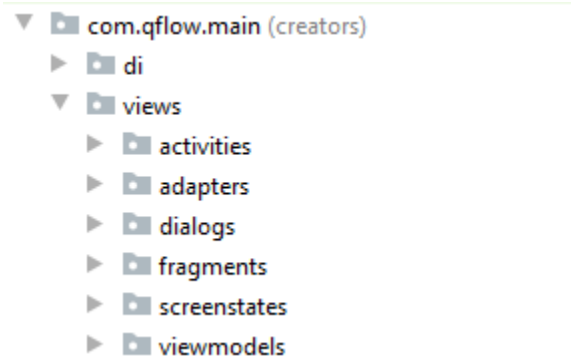


Figura 6.6 Estructura Flavours

En la Figura 6.6 se puede observar la estructura que va a seguir cada uno de los *Flavours*, dejando ver que las partes que se van a tener que implementar para cada una de las aplicaciones realmente no va a ser tanto.

Cada uno de los paquetes se explica bien por sí mismo, pero hay dos que requieren algo de atención. Estos son los de *Adapters*, los cuales guardan los adaptadores de las *Recycler Views* usadas, y los *Screenstates*, que almacenan los estados que pueden tener cada una de las vistas.

6.3 Funcionalidades de la aplicación

Con la arquitectura bien definida e implementada continuamos con la siguiente parte del proceso del desarrollo, la definición y el desarrollo de las diferentes funcionalidades de la aplicación.

A continuación, se muestran los primeros bocetos de las vistas principales y las funcionalidades de la aplicación, explicando cómo estas se han implementado. Se han explicado las funcionalidades dependiendo de la aplicación que se esté ejecutando, en primer lugar, las funcionalidades comunes a administradores y usuarios, luego las propias a la aplicación de un usuario sin privilegios y por último las exclusivas a los administradores de colas.

6.3.1 Bocetado de las vistas principales

En el desarrollo, se ha decidido reducir al máximo el tiempo dedicado al diseño de las vistas de las aplicaciones. Esta decisión ha sido tomada gracias a que Android Studio ofrece la posibilidad de realizar maquetaciones en directo, que nos han ahorrado el tiempo de realizar maquetas en otros softwares.

A pesar de esto, a medida que ha ido avanzando el desarrollo, podría existir la necesidad de realizar un nuevo maquetado con el fin de revisar las vistas ya creadas.

Sin embargo, aun existiendo esta falta de maquetación, para que el equipo tuviese una idea común de cara al desarrollo se realizaron los bocetos de las vistas principales mostradas en la Figura 6.7 y Figura 6.8.

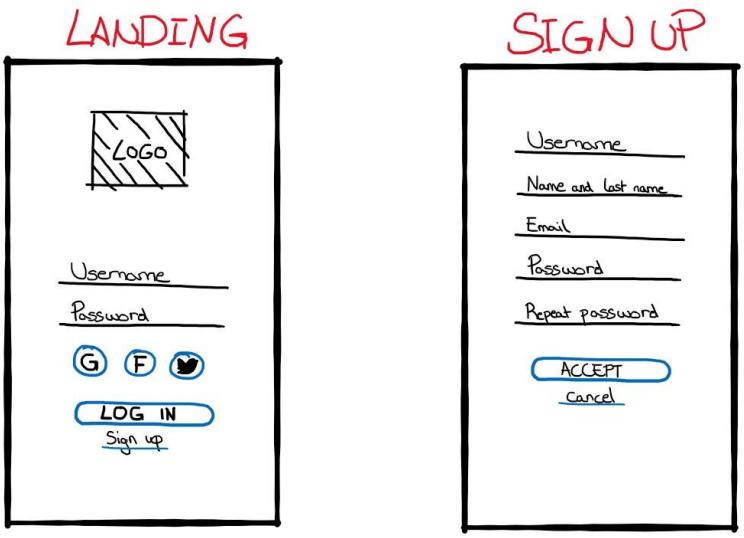


Figura 6.7 Bocetos de *Landing* y *SignUp*

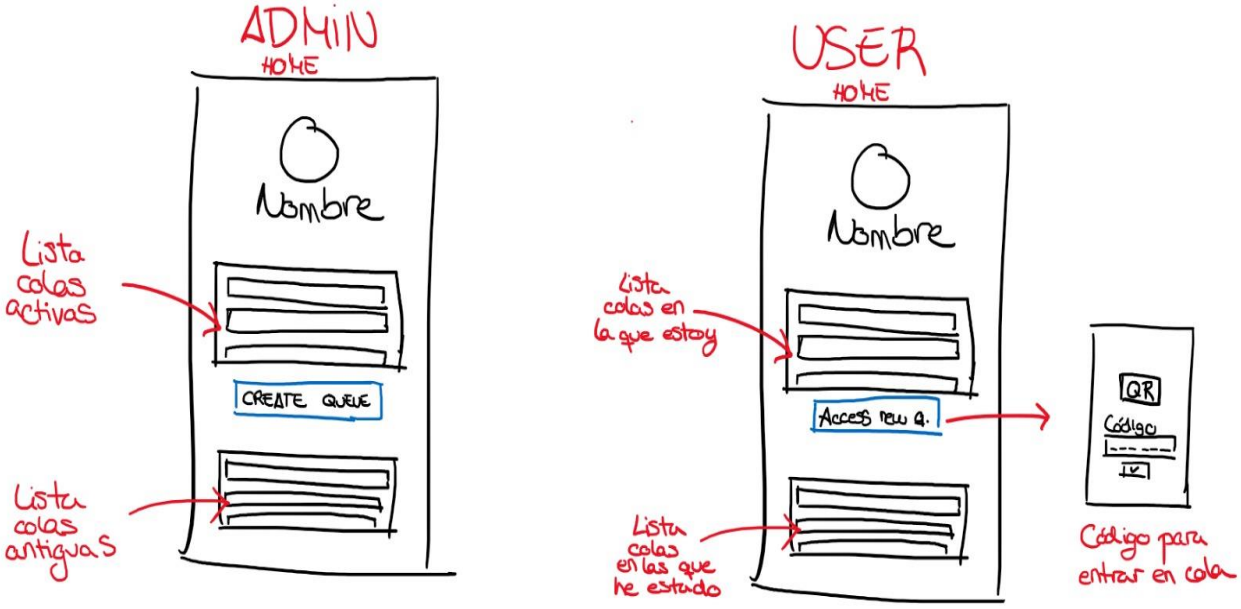


Figura 6.8 Bocetos vistas Home

6.3.2 Funcionalidades comunes a usuarios y administradores

Estas funcionalidades se repiten tanto a nivel usuario como a nivel creador difiriendo solo en el diseño de la interfaz.

Las tres únicas funcionalidades de nuestra aplicación que cumplen este criterio son *SignIn* que permite a los usuarios acceder a su cuenta, *SignUp* es el formulario de registro para que un usuario se de alta y la codificación de la contraseña en MD5 [39].

Cómo en cualquier aplicación de hoy en día, el *SignIn* es usado para acceder a la aplicación siendo un usuario normal o administrador. Dependiendo en qué aplicación se esté, se envía al servidor la señal de si el *Login* ha sido realizado por un administrador o un usuario sin privilegios.

Esta llamada se realiza recogiendo el email y la contraseña del usuario, que se envía al servidor con las variables *isAdmin* que indica si es creador o no a través del *ApiService*. Si el servidor devuelve el Json de forma correcta, se añade el *token* del usuario al repositorio *SharedPreferences*, usado durante toda la sesión del usuario actual. Tras ello se cambia el *Screenstate* para actualizar la vista, que pasa al *Home* donde el usuario logueado puede ver las colas actuales en las que está, el historial de colas visitadas y opciones como unirse a cola, etc.

SignUp es la funcionalidad que permite a un usuario o creador crear su cuenta en nuestro servicio. Esta cuenta se crea con las capacidades de un creador o un usuario común, como pasa con el *SignIn*, dependiendo de la aplicación que se esté usando para realizar el registro.

Los campos que un usuario o creador tienen que rellenar son los mismos: nombre de usuario, nombre y apellido, email, contraseña y repetir contraseña.

Después de rellenar estos datos, se envía al servidor a través del *ApiService* en un objeto *POST*. Tras ser confirmada la inserción del usuario a la base de datos en la se recibe un objeto Json con la información básica del usuario creado.

Las funcionalidades de *SignIn* y *SignUp* se pueden ver implementadas en la Figura 6.9.

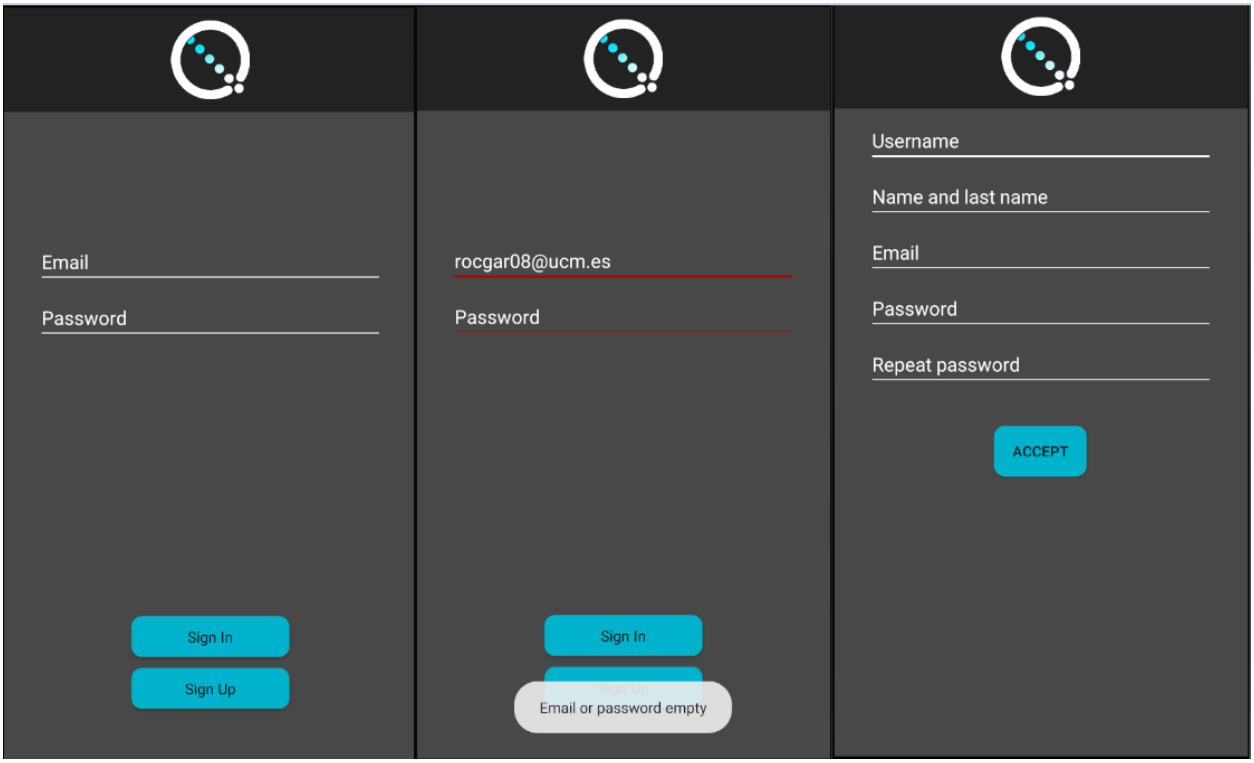


Figura 6.9 Funcionalidades comunes

Al realizar esta confirmación se cambia el *ScreenStates* y el usuario es dirigido a la pantalla *Home.*

Para evitar que las contraseñas se guarden en texto plano, se ha implementado una clase que permite codificarlas mediante el algoritmo *hash MD5* de manera provisional.

6.3.3 Funcionalidades específicas de la aplicación de usuarios

En los siguientes apartados se han detallas las funcionalidades correspondientes a los usuarios de la aplicación que no tienen permisos de administrador. Estas funcionalidades son exclusivamente accesibles a través de la aplicación de un usuario sin privilegios.

Para conseguir la información asociada a las colas se consulta al servidor mediante el *ApiService* en un objeto *GET* que devuelve el nombre, la capacidad de la cola y si la cola está bloqueada o no.

Desde la vista de *Home*, el usuario a través del botón de *Join* accede a la pantalla de unirse a una cola, donde introduce el código asociado a esta cola. Tras esto, se muestran el listado de colas a las que se ha apuntado mostrando el nombre de la misma, si es su turno o cuánto tiempo le toca esperar. Junto a estas listas hay un botón que muestra la información asociada de la cola. En esta misma pantalla también se muestra un listado de las colas a las que ha pertenecido. Todo este flujo se muestra en la Figura 6.10.

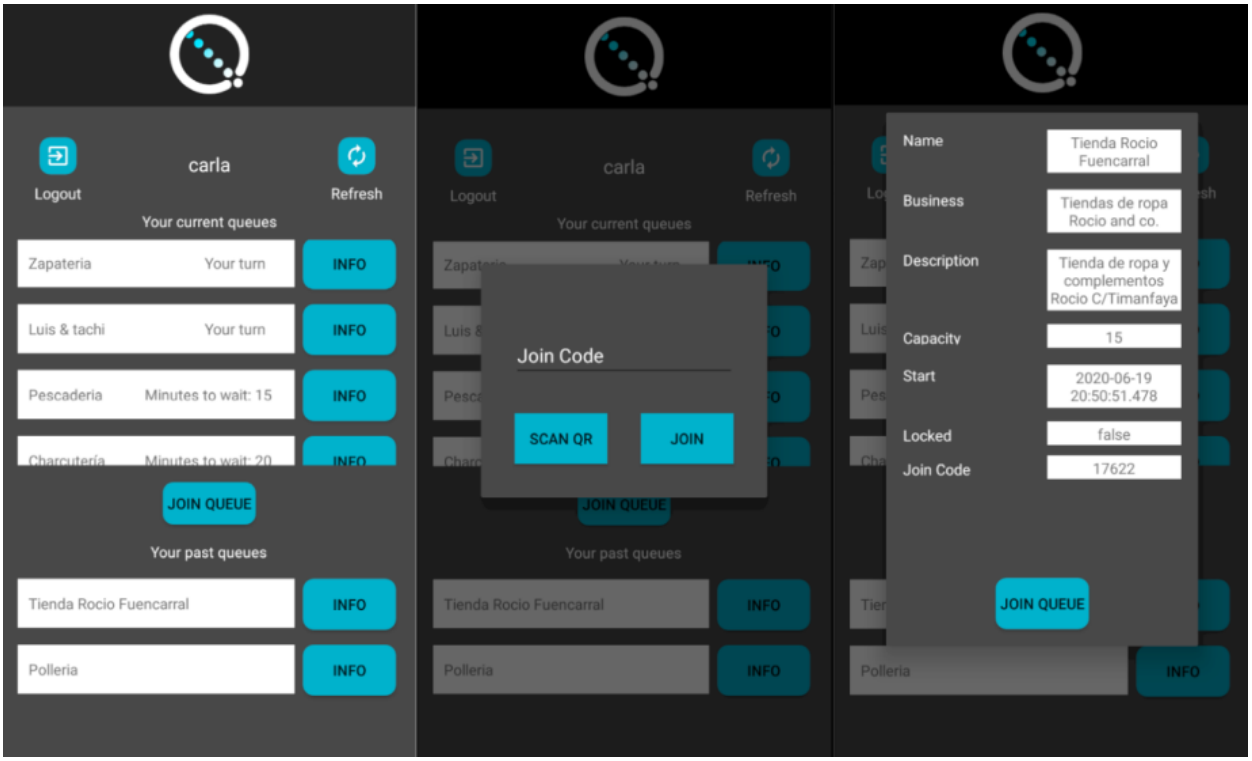


Figura 6.10 Funcionalidades de los usuarios

Cuando el usuario inserta el código, este dato es enviado al servidor mediante el *ApiService* en un objeto *POST*. Una vez hechas las comprobaciones en el servidor se añade al usuario en la cola y mediante el *ScreenStates* vuelve a su vista principal.

Además de la funcionalidad anterior, un usuario también se puede unir a una cola escaneando un QR a través de su cámara tras haber dado los permisos necesarios como se muestra en la Figura 6.11.

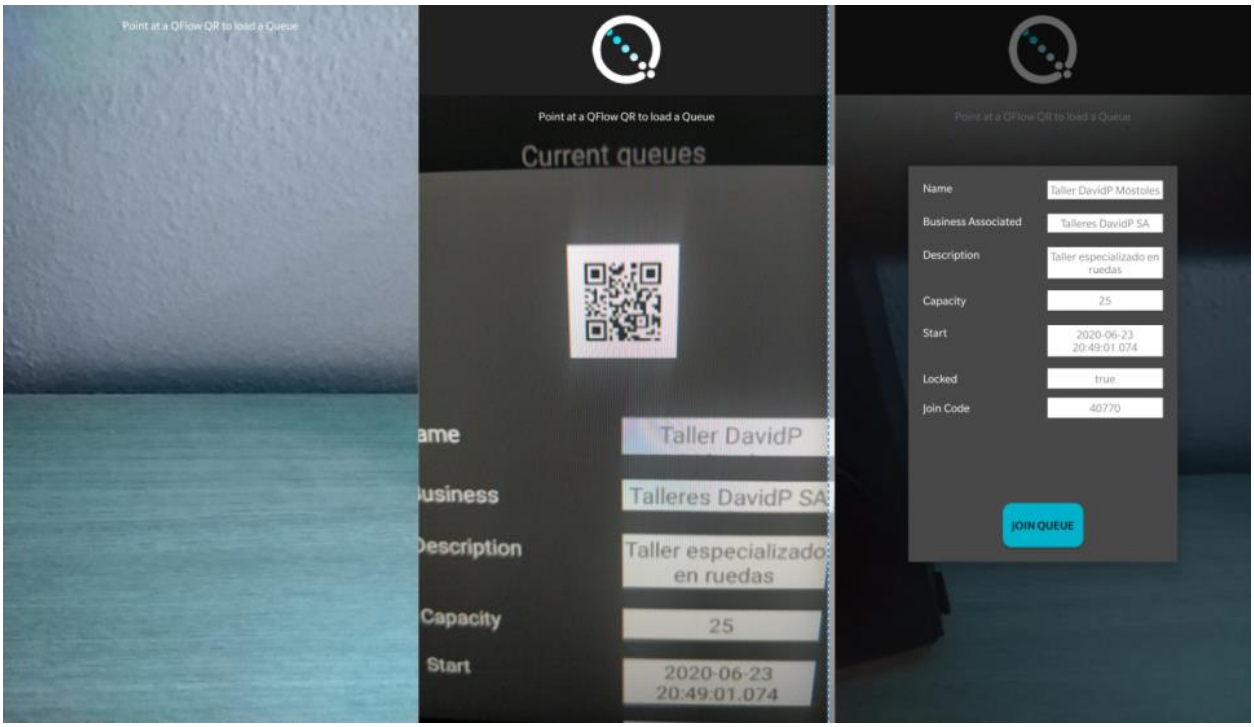


Figura 6.11 Flujo de Lectura QR

Una vez realizadas las comprobaciones en el servidor se añade al usuario en la cola y mediante el *screenstate* el usuario vuelve a la vista principal.

En la vista principal se muestran las colas a las que ha pertenecido el usuario, esta información se consulta al servidor mediante el *ApiService* en un objeto *GET* devolviendo el nombre de las colas que han sido asociadas a ese usuario con anterioridad.

Concretamente, la información que se muestra de las colas mientras estén activas es: cuánto tiempo de espera se estima que queda, cuándo es tu turno, el turno y finalmente una vez pasa tu turno se mueve a la lista de colas no activas.

6.3.4 Funcionalidades específicas de la aplicación de creadores

Las funcionalidades descritas a continuación son únicas para los administradores de colas que se pueden encargar de las distintas gestiones de una cola.

El *Home* del creador muestra las colas creadas por sí mismo, en la primera lista se muestran aquellas que están activas, mientras que en la segunda se encuentran aquellas ya cerradas.

La funcionalidad de crear una cola es aquella en la que un administrador da de alta una cola rellenando los campos de nombre, negocio, descripción, capacidad, fecha de inicio, fecha de finalización, si la cola está bloqueada o no y el código de unirse a una cola. Tras rellenar los datos se envía al servidor mediante el *ApiService.kt* en un objeto Json. Una vez confirmada la inserción de la cola a la base de datos se recibe un objeto Json con la información básica de la cola creada. Después de hacer esta confirmación se cambia el *screenstate* y el administrador es dirigido de nuevo a la pantalla Home, para observar este flujo ver Figura 6.12. Una vez introducidos los datos correctamente a través del *screenstate* se lleva al administrador a la vista de *Home.*

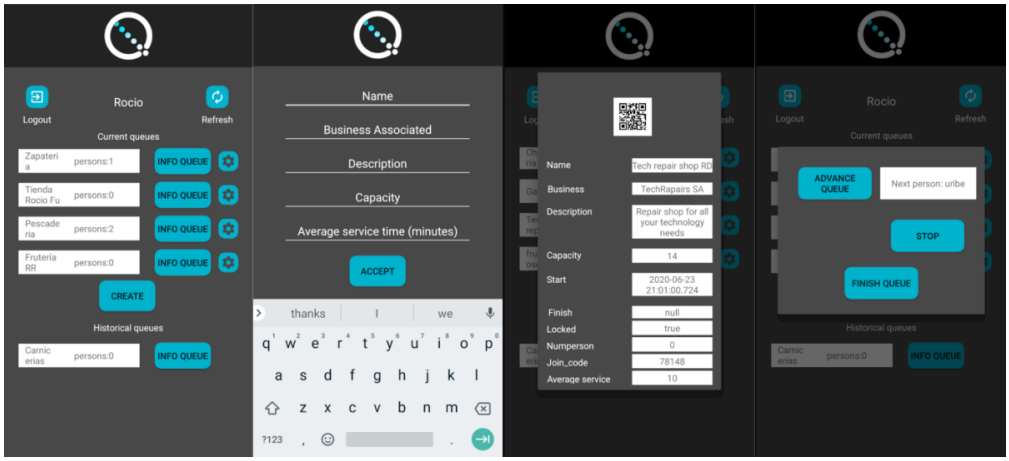


Figura 6.12 Funcionalidades principales del creador

A continuación, se detallan las funcionalidades que se encargan de gestionar las colas.

Avanzar una cola es en la que el administrador selecciona la cola a modificar, mediante el *screenstate* navega hacia el panel de control donde al pulsar el botón de avanzar se realiza una llamada al servidor mediante el *ApiService* en un objeto *POST* que devuelve la posición actual de la cola, el nombre del usuario que le toca avanzar saliendo así de la cola y otra información relacionada con ella. Tras actualizar la posición actual de la cola mediante el *screenstate* el administrador vuelve a la pantalla principal.

La funcionalidad de parar una cola se realiza desde la vista de *Home*. El administrador pulsa sobre la cola a modificar y mediante el *screenstate* va al panel de control donde al pulsar sobre el botón *parar*, éste llame al servidor mediante el *ApiService* en un objeto *POST* que devuelve el estado actual de la cola. Una vez actualizado el estado de la cola el administrador vuelve a la pantalla principal.

Reanudar cola es en la que el administrador desde su vista principal selecciona la cola a modificar y mediante el *screenstate* navega al panel de control donde al pulsar el botón reanudar se hace una llamada al servidor mediante el *ApiService* en un objeto *POST* que devuelve el estado actual de la cola. Una vez realizada esta modificación el usuario vuelve a su vista principal.

Cerrar la cola ocurre cuando un administrador selecciona la cola que quiere desactivar. Mediante el *screenstate* se envía al administrador al panel de control. Al darle al botón de cerrar se realiza una llamada al servidor mediante el *ApiService* en un objeto *Post* devuelve el estado de la cola con la fecha de finalización del evento. Después de actualizar este campo mediante el *screenstate* el administrador vuelve a la pantalla principal y se añade la cola cerrada a la vista del historial de colas.

En la vista *Home*, del administrador se muestra la lista con el nombre de las colas actuales creadas por este. Para conseguirla realiza una llamada al servidor mediante el *ApiService* en un objeto *POST* que devuelve la información de la cola. Además, mediante el *screenstate* se pasa al administrador a otra pantalla desde donde puede editar los campos de la cola.

Por último, en la vista principal se muestran las colas creadas con anterioridad por el administrador, esta información se consulta al servidor mediante el *ApiService* en un objeto *GET* que devuelve el nombre de las colas creadas con anterioridad por el administrador.

Capítulo 7 - Servicio y despliegue

En este capítulo se explicará todo lo relacionado con el servidor que respalda a la aplicación, junto a todo lo necesario para su funcionamiento online. Comenzaremos explicando el desarrollo en Spring, seguido por las funcionalidades implementadas. En el Anexo 3 se explica el despliegue de Heroku.

7.1 Diseño del servidor en Spring

Originalmente, como se ha mencionado anteriormente, Google Firebase iba a ser utilizado para realizar las funciones, pero finalmente se decidió cambiar a un servicio desarrollado por el equipo.

Esta decisión se tomó debido a la inexperiencia del equipo en cuanto al desarrollo en NodeJS sumado a la necesidad de tener una base de datos relacional que permitiese búsquedas relativamente complejas, de manera que se redujese la carga operativa en la gestión de colas.

Habiendo abandonado Firebase en una situación bastante avanzada del proyecto, era necesaria una tecnología que no tuviese una gran curva de aprendizaje y que permitiese hacer todo lo que no nos permitía la tecnología anterior.

Por ello elegimos Spring ya que, al tener una base en Java, el equipo estaba acostumbrado al desarrollo. Además, aquello que añadía de específico esta tecnología no aportaba demasiada complejidad al proyecto.

7.1.1 Arquitectura del servidor

A diferencia de Firebase, el servidor en Spring es un proyecto de Java, lo que conlleva un desarrollo similar al que tenemos en la aplicación de Android. Es decir, tiene que haber una arquitectura correcta de manera que se cumplan los criterios *SOLID*.

Por ello, el patrón más utilizado en los proyectos de Spring, y que se aplicará en el proyecto, ha sido el patrón Modelo Vista Controlador [40] modificado para aplicar la Arquitectura *Clean*, de una manera similar a como lo hacemos en la aplicación. La Figura 7.1 representa el funcionamiento de la arquitectura.

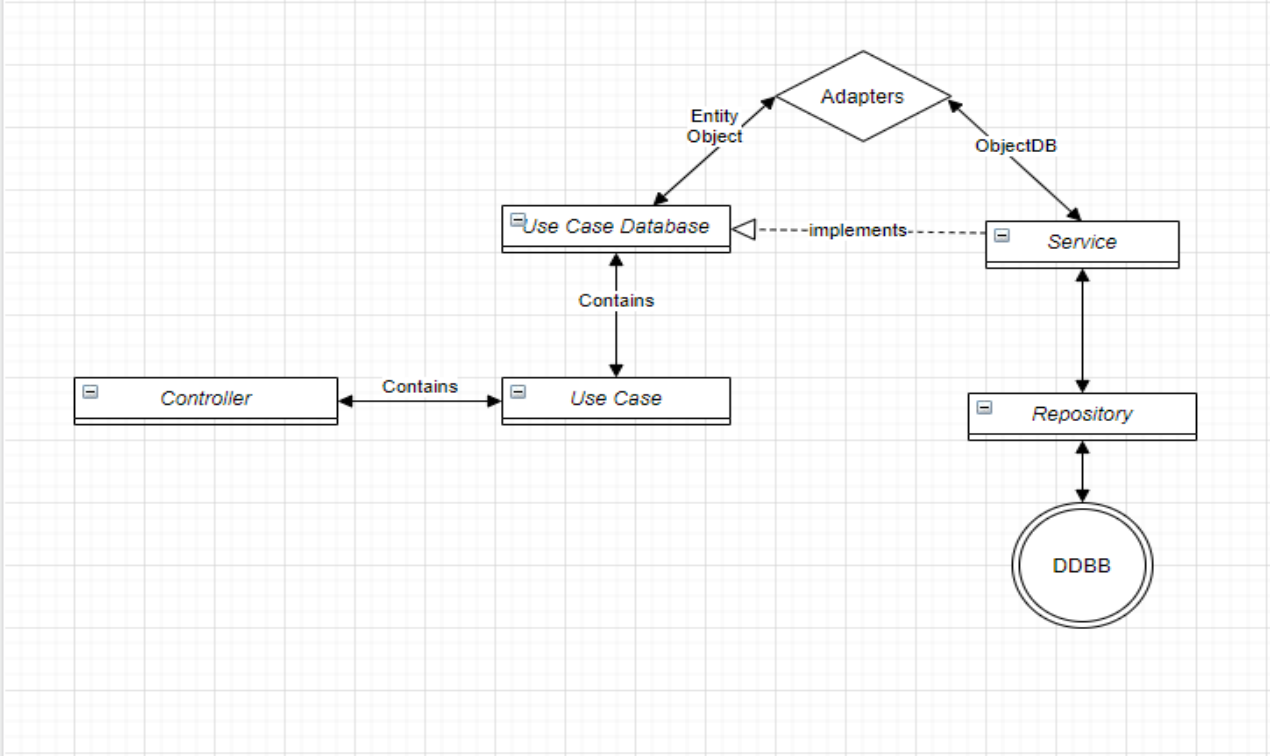


Figura 7.1 Arquitectura final del servicio en Spring

Por tanto, en el servicio tendremos los siguientes elementos:

- Controladores, encargados tanto de recibir las peticiones, junto con los datos que estas precisen, como de devolver el resultado una vez terminada la ejecución.
- Los *Use Cases*, los cuales al igual que en Android, realizan toda la lógica necesaria para llevar a cabo una funcionalidad. Sin embargo, a diferencia de la implementación en la aplicación, aquí contienen una clase *Database* de los mismos. Estas son interfaces que se encargan de hacer toda la lógica relacionada con la Base de Datos, y que es implementada por los servicios.
- Servicios, heredan de estos *Use Cases Database* e implementan todo lo relacionado con la conexión de los repositorios.
- Repositorios, hacen toda la lógica relacionada con el funcionamiento típico de la base de la base de datos, es decir, insertar, eliminar, actualizar, etc.

Es importante diferenciar que el servicio cuenta con dos tipos de objetos, de cara a una buena modularización: Aquellos definidos en el paquete *Entitiy*, que son aquellos que se utilizan en los Use Cases, y los definidos en el paquete DTO de los repositorios, diferenciados por el sufijo DB en su nombre, que se usan solo en los repositorios y servicios. Los adaptadores serán los que se encargan de realizar las conversiones entre estos

En lo referente a tecnologías utilizadas en el servidor, Spring ofrece de base prácticamente todo lo necesario, por lo que la preparación del proyecto fue más sencilla que en Android. A pesar de esto, se han tenido que introducir ciertos elementos que merecen ser explicados.

Por un lado, la inyección de dependencias con *Bean* [41] ya que, al utilizar una arquitectura como la anterior, se tiende a separar la funcionalidad en diferentes clases muy especializadas, lo que conlleva tener una gran cantidad de clases que pueden llegar a repetirse en tiempo de ejecución. Para ello, como en Android, se ha decidido usar un inyector de dependencias de manera que se creen instancias únicas.

Por otro lado, la carga de la Base de Datos en el servicio, la cual se realiza con *Flyway* [42], que permite cargar desde diferentes clientes de bases de datos, como es nuestro caso con PostGreSQL, a partir de las instrucciones de creación de tablas desde un fichero .sql.

A pesar de intentar modularizar en todo lo posible la implementación de las funcionalidades del servicio, todas coinciden en que pasan por un controlador y un servicio común relacionado con la funcionalidad que traten (colas y usuarios).

Esto, sumado a que ahora para probar las funcionalidades necesitamos lanzar el servicio y hacer pruebas mediante Postman, nos lleva a la necesidad de realizar *test* para probar si estas partes más conflictivas funcionan correctamente antes de realizar despliegues. Por ello se ha tomado la decisión de realizar un desarrollo orientado a *test* [43] de cara a los controladores y los servicios. Es decir, primero se definen los *test* y después se realiza la implementación de la lógica del servicio. En total se han realizado 41 test que como se observa en la Figura 7.2 han pasado las pruebas definidas con anterioridad.

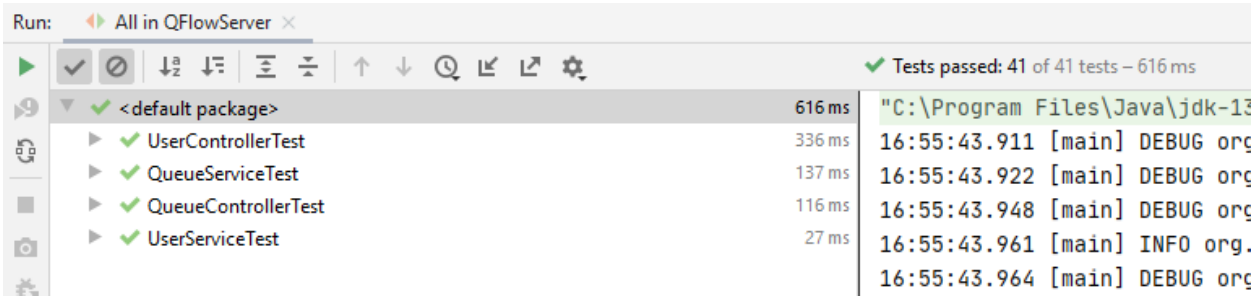


Figura 7.2 Captura Test

Para llevar esto a cabo este proceso se ha utilizado la tecnología de Mockito [44] a la hora de realizar *mocks* y JUnit [45] para ejecutar y crear los *test*.

7.1.2 Estructura final del proyecto

Finalmente conociendo la arquitectura que se ha utilizado junto con las tecnologías, la estructura va a ser la mostrada en la Figura 7.3.

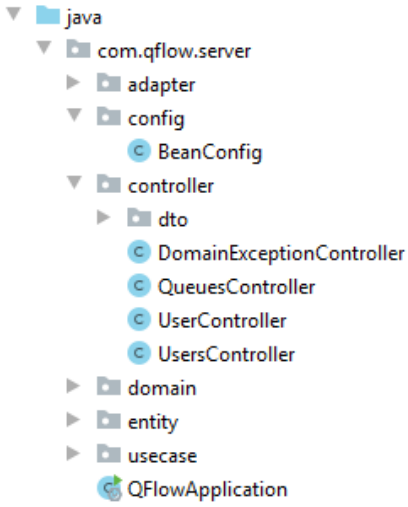


Figura 7.3 Estructura del proyecto en carpetas

- *Adapters*, contiene los adaptadores que convierten las clases entre los diferentes objetos del modelo.
- *Config*, contiene la configuración del inyector de dependencias.
- *Controller*, contiene los distintos controladores, incluyendo el de excepciones.
- *Domain*, contiene los paquetes de los servicios y los repositorios, es decir, todo lo relacionado con la conexión y ejecución cercana a la Base de Datos.
- *Entity*, guarda las clases que representan a los objetos de la Base de Datos de cara a la ejecución en el servidor.
- *Use Case*, contiene los *Use Cases* junto a los *UseCasesDatabase.*
- *QFlowApplication*, el cual es el punto de entrada del servicio.

7.2 Funcionalidades del Servidor

Estas funcionalidades desarrolladas en el servidor se usan para gestionar las colas a distintos niveles. Lo más básico es crear, unir y listar colas.

Las funcionalidades referidas al usuario constan de llamadas *POST*, *PUT* y *GET* al servidor, siendo las más básicas registrar a un usuario, conseguir información del usuario e iniciar sesión de un usuario.

Para documentar y testear nuestra API se ha hecho a través de Swagger Inspector y Swagger Hub. Con el primero se probaron los *endpoints* y generamos la definición de la API y con el segundo se mostraron las definiciones de la API, los parámetros, las cabeceras y los objetos Json.

7.2.1 CreateQueue

Llamada que se realiza cuando se crea una cola desde la aplicación, acción que solo puede realizar un administrador desde la aplicación Qflow Creators.

Para la creación de una cola, se realiza una llamada del tipo *POST*, en la que llegan al servidor los parámetros *token* y un objeto Json, como se muestra en la Figura 7.4, con todos los datos requeridos para poder guardar la cola en la base de datos. Estos datos no son todos los que contiene la cola en la base de datos ya que algunos se generan de forma automáticamente en el servidor: id, fecha de creación, posición actual, y su estado de bloqueo.

La información recibida desde la llamada junto a la generada en el servidor se guarda en la base de datos en la tabla *queue* a través de un objeto *QueueDB* y la necesaria en la tabla *queue_user* con un objeto *QueueUserDB*, que contiene las relaciones usuario y cola, siendo en este caso la relación de administrador.

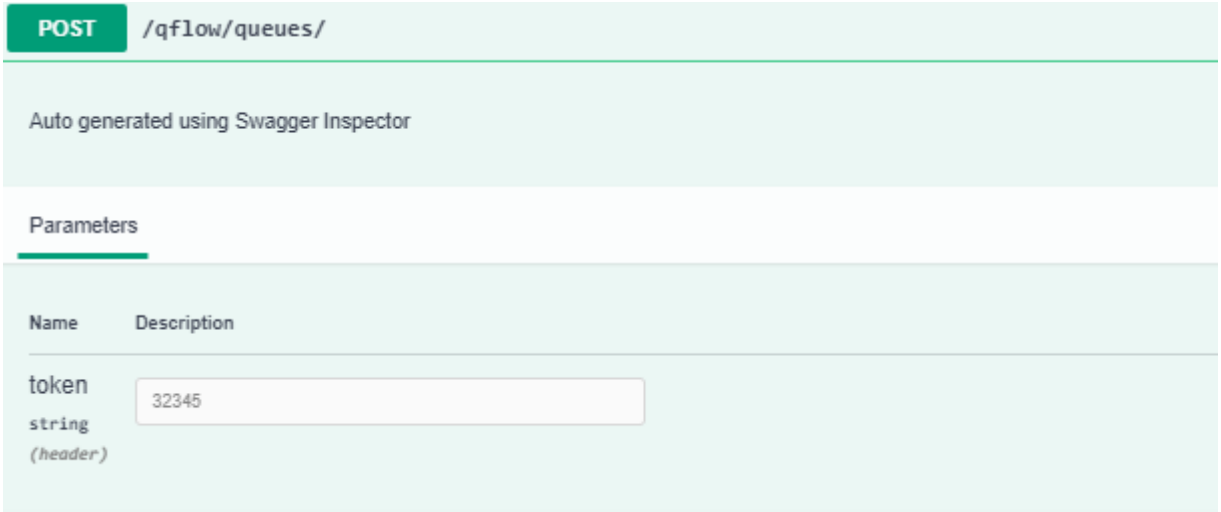


Figura 7.4 Definición Swagger CreateQueue

Para crear las colas se usa un objeto Json como se muestra en el siguiente ejemplo

"name": "cola swagger",
"description": "Prueba swagger",
"dateCreated": "2020-27-23T17:31:05.000+0000",
"dateFinished": "2020-28-23T17:31:05.000+0000",
"capacity": 300,
"currentPos": 1,
"isLock": false,
"businessAssociated": "swaggerorg"

7.2.2 *JoinQueue*

Esta funcionalidad se usa cuando un usuario después de introducir el código de la cola acepta el unirse a esta cola, añadiéndose la información a la base de datos.

En la llamada al servidor con esta funcionalidad, de tipo *POST*, como se muestra en la Figura 7.5, recibimos el parámetro el *join_id* de la cola y el *token* del usuario que se quiere unir. Con estos dos datos se accede a los datos de la cola, y así se verifica en el servidor que el usuario no esté en esa cola y que la capacidad de la cola no ha llegado a su límite. Si se cumplen las condiciones, se incluye al usuario dentro de la cola, es decir, en la tabla *queue_user*, con un objeto *QueueUserDB* con su posición dentro de la cola y se actualizan las tablas de las relaciones correspondientes, en este caso *info_queue_user* a través de un objeto *InfoQueueUserDB*. En el constructor de éste, se genera la información de la fecha de entrada y la variable que indica si el cliente abandona la cola en algún momento, que se inicializa a *false*.

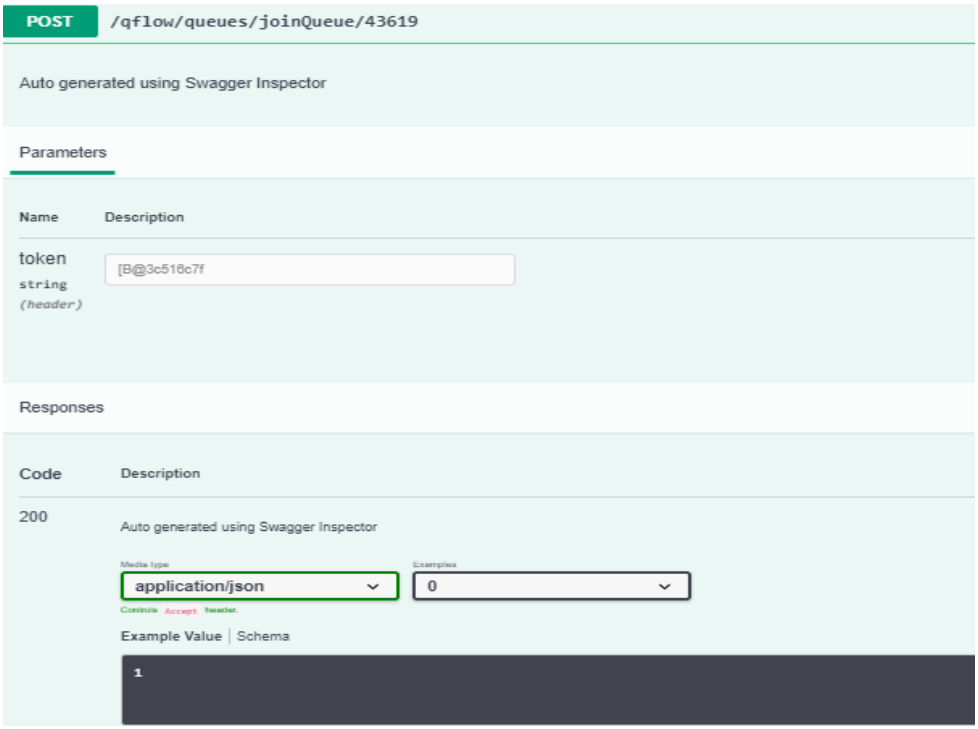


Figura 7.5 Definición Swagger JoinQueue

7.2.3 GetQueues

A la hora de conseguir las colas de la base de datos diferenciamos entre dos llamadas, ambas del tipo *GET*.

La primera es *getQueueByQueueId* que simplemente recibe el *id* de la cola a buscar y devuelve su información en un objeto *Queue* con toda la información de ésta, obtenida a través de una llamada a *findById* en la base de datos, tal como se muestra en la Figura 7.6.

La segunda llamada es algo más compleja es *getQueueByUserId*. Ésta se usa para conseguir las colas que están relacionadas con un usuario pudiendo ser esta relación la de administración o la de espera en la cola, actualmente o en el pasado. También se usa para conseguir todas las colas de la base de datos en general.

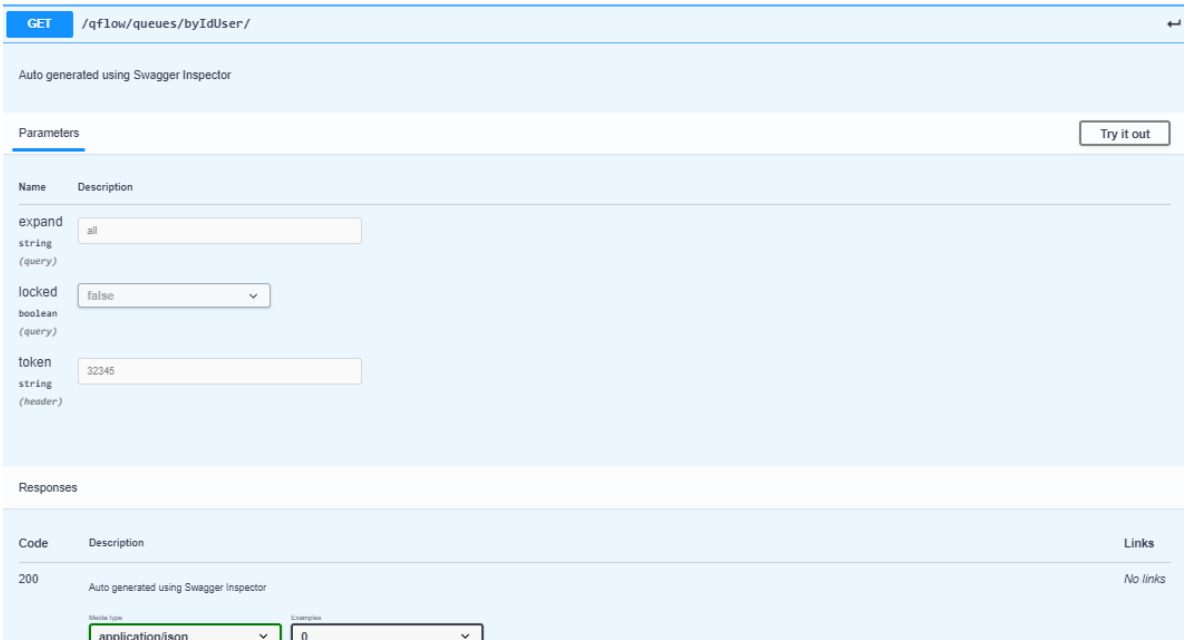


Figura 7.6 Definición Swagger getQueueByIdUserId

Para diferenciar entre estos usos se añade a la llamada al servidor un parámetro *finished* que indica si las colas que se buscan están activas o han quedado en el pasado, y un parámetro *expand*, que permite llamarlo con valores como *user*, *creator* o *all* para distinguir entre un usuario, un administrador u obtener la información de todas las colas en la base de datos, sin importar su estado. Este parámetro *expand* nos da flexibilidad para poder añadir más usos a esa misma función, solo teniendo que añadir *queries* nuevas ver Figura 7.6.

Como se ha comentado en el capítulo de teoría de colas, el tiempo medio de espera de un cliente en la cola se calcularía mediante la distribución Log-Normal.

Una vez realizada la consulta *GET* nos devuelve un documento Json con las colas asociadas a ese usuario como se muestra a continuación.

[{"id":6,"name":"bobo","description":"bob3","joinId":96842,"dateCreated":"2020-05-23T12:11:48.202+0000","dateFinished":"2020-05-23T18:03:53.000+0000","capacity":12,"currentPos":1,"businessAssociated":"sony","lock":true,“nextPerson”:”MariaDF”},{id":3,"name":"Queue by hand b","description":"Queue by hand for testing","joinId":12343,"dateCreated":"2020-05-23T17:31:06.000+0000","dateFinished":null,"capacity":200,"currentPos":1,"businessAssociated":"Test business","lock":false,“nextPerson”:”JuanJM”},{"id":1,"name":"Test","description":"Queue Test","joinId":43619,"dateCreated":"2020-05-23T17:31:02.000+0000","dateFinished":null,"capacity":300,"currentPos":1,"businessAssociated":"Test business","lock":false,“nextPerson”:”MariaDF”}]

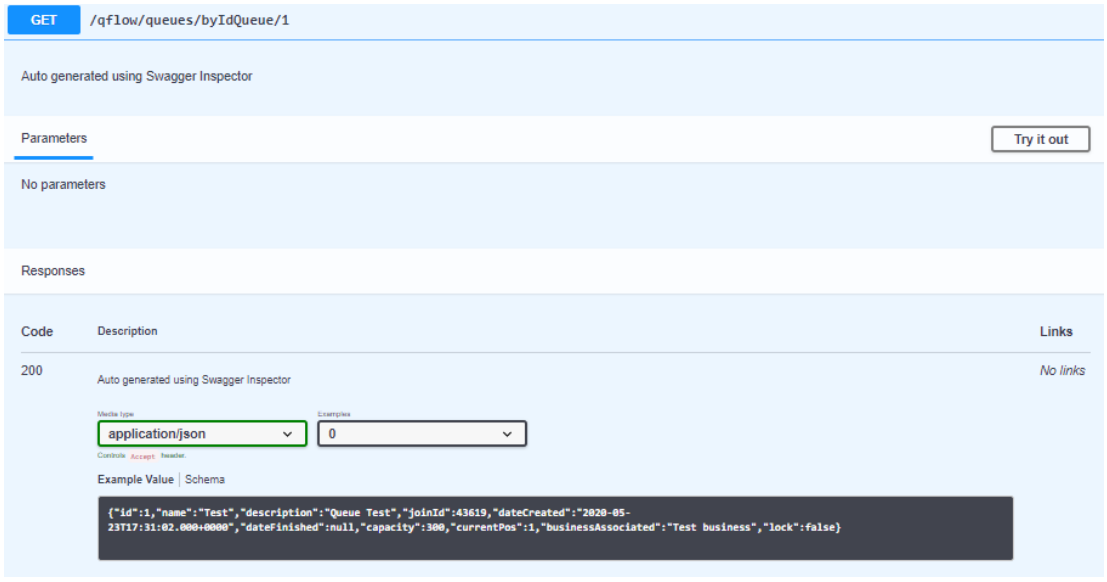


Figura 7.7 Definición Swagger getQueueByQueueId

Al hacer esta llamada devuelve un Json con la información de la cola

{"id":1,"name":"Test","description":"Queue Test","joinId":43619,"dateCreated":"2020-05-23T17:31:02.000+0000","dateFinished":null,"capacity":300,"currentPos":1,"businessAssociated":"Test business","lock":false}

7.2.4 Avanzar cola

Avanzamos a un usuario en la cola a través de una llamada *POST*, el servidor recibe el *id* de la cola y el *token* relacionado con el usuario asociado a la cola como se muestra en la Figura 7.8.

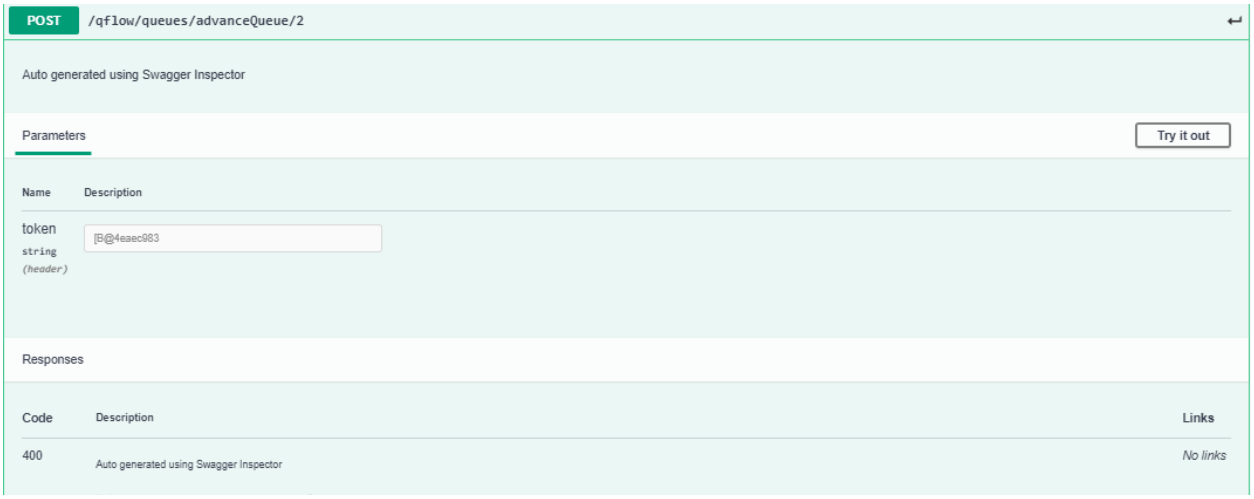


Figura 7.8 Definición Swagger avanzar cola

El servidor se encarga de cambiar el valor de la posición que ocupa el usuario en la cola actual, además de actualizar los valores del usuario en la cola, devolviendo un Json con los datos de la cola y actualizada la posición actual del usuario.

{"id":2,"name":"Queue by hand","description":"Queue by hand for testing","joinId":43019,"dateCreated":"2020-05-23T17:31:05.000+0000","dateFinished":"2020-06-14T17:34:08.610+0000","capacity":140,"currentPos":4,"businessAssociated":"Test business","numPersons":1,"inFrontOfUser":0,"avgServiceTime":0,"waitingTimeForUser":0,"lock":false}

7.2.5 Parar cola

Al parar una cola a través de una llamada *POST*, el servidor recibe el *id* de la cola que quiere parar el administrador como se muestra en la Figura 7.9. El servidor se encarga de cambiar el valor *locked* de esa cola y volverla a guardar en la base de datos. El período de actividad de la cola se actualiza en la tabla *active_period*.

Figura 7.9 Definición Swagger parar cola

Esta llamada al servidor devuelve un Json con la información de la cola y se actualiza el estado de *isLock* a *true*.

{"id":2,"name":"Queue by hand","description":"Queue by hand for testing","joinId":43019,"dateCreated":"2020-05-23T17:31:05.000+0000","dateFinished":null,"capacity":140,"currentPos":1,"businessAssociated":"Test }business","numPersons":0,"inFrontOfUser":0,"avgServiceTime":0,"waitingTimeForUser":0,"lock":true}

7.2.6 Reanudar cola

Al volver a poner en marcha una cola a través de una llamada *POST*, el servidor recibe el *id* de la cola que quiere reanudar el administrador como se muestra en la Figura 7.10. El servidor se encarga de cambiar el valor *locked* de esa cola y volverla a guardar en la base de datos además de actualizar el periodo de actividad de la cola en la tabla *active_period*. En caso de que no exista un valor previo, se crea uno nuevo y lo guarda.

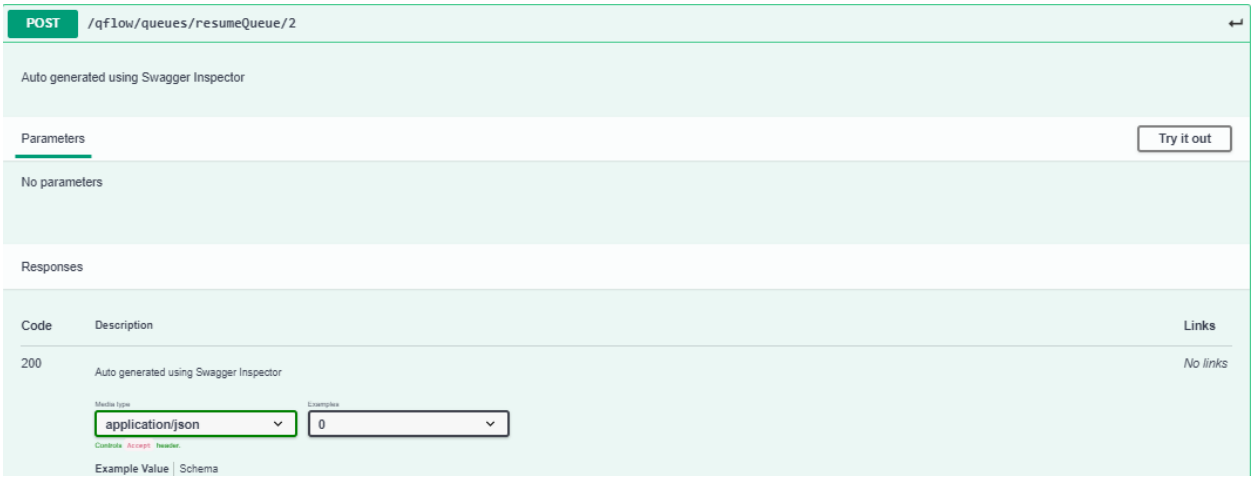


Figura 7.10 Definición Swagger reanudar cola

Como resultado, devuelve un Json con la información de la cola y cambiando el estado de *isLock* a *false*.

{"id":2,"name":"Queue by hand","description":"Queue by hand for testing","joinId":43019,"dateCreated":"2020-05-23T17:31:05.000+0000","dateFinished":null,"capacity":140,"currentPos":1,"businessAssociated":"Test business","numPersons":0,"inFrontOfUser":0,"avgServiceTime":0,"waitingTimeForUser":0,"lock":false}

7.2.7 Cerrar cola

Al cerrar una cola a través de una llamada *POST*, el servidor recibe el *id* de la cola que quiere cerrar el administrador como se muestra en la Figura 7.11. El servidor se encarga de cambiar el valor *locked* de esa cola, el valor de *date_finished* a la fecha actual y volverla a guardar en la base de datos y se actualiza el periodo de actividad de la cola en la tabla *active_period*.

Figura 7.11 Definición Swagger cerrar cola

7.2.8 CreateUser

Al registrar a un usuario a través de una llamada *POST*, el servidor recibe un objeto *UserPost* a través de un Json que se trata y se guarda en la base de datos. Este usuario será administrador o no dependiendo de la aplicación desde la que realice el registro asignado por el campo *isAdmin* ver Figura 7.12.

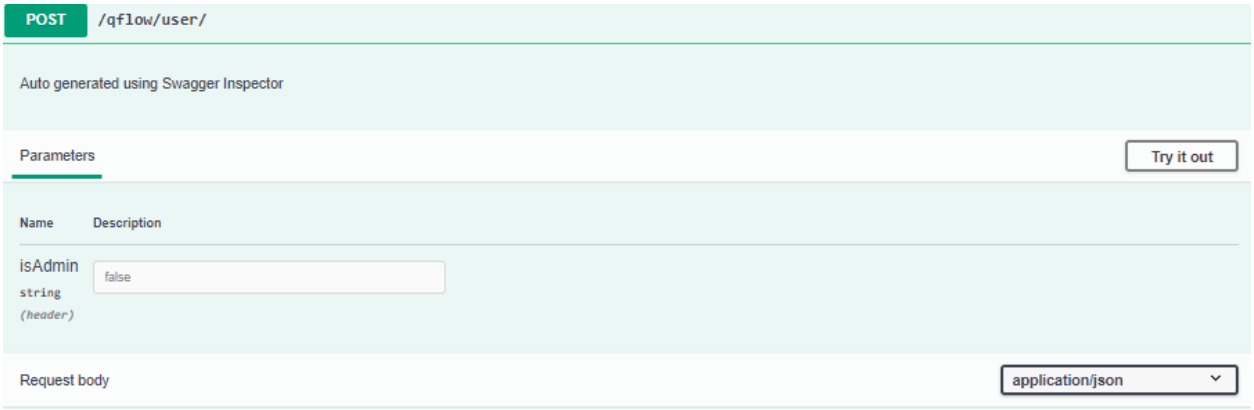


Figura 7.12 Definición Swagger CreateUser

A continuación, se muestra el objeto Json con los campos correspondientes para crear el usuario

```
"id": null,
"email": "pruebaro@ro",
"name_lastname": "pruebaro",
"password": "12345",
"profile_picture": "",
"username":"prueba"
```

7.2.9 LoginUser

Cuando un usuario inicia sesión la llamada en el servidor recibe los parámetros necesarios en una llamada de tipo *PUT*: mail, contraseña y el valor de *isAdmin*, que marca si un usuario es un administrador o no dependiendo de la aplicación desde la que realicen el inicio de sesión.

Se busca a este usuario en la base de datos teniendo en cuenta el parámetro *isAdmin* y si éste existe se devuelve un objeto *UserDTO* con la información resumida de ese usuario, este objeto contiene nombre de usuario, email y *token* como se muestra en la Figura 7.13.

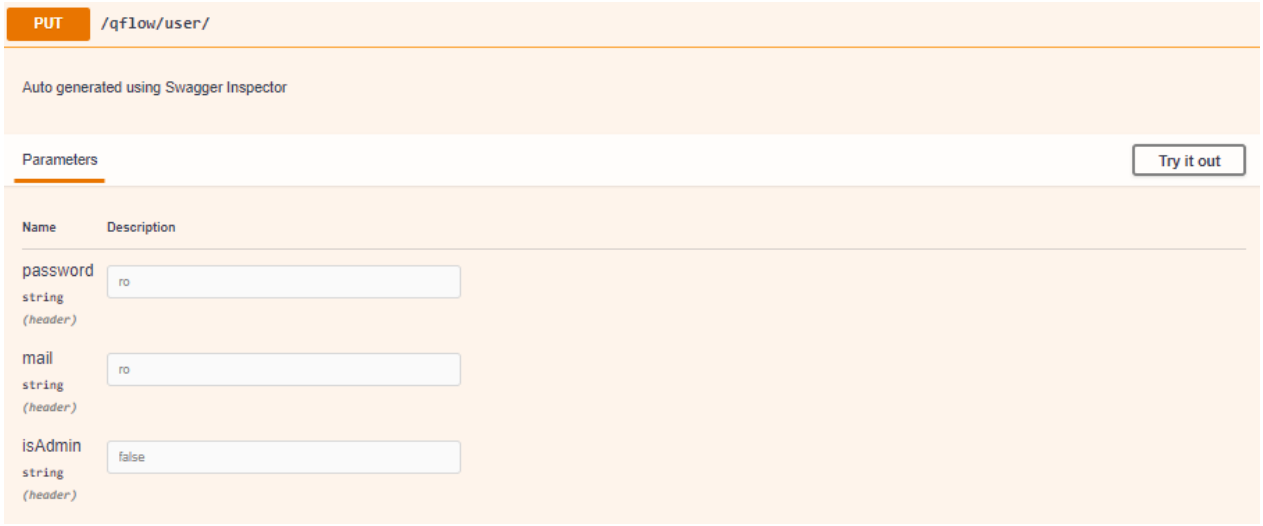


Figura 7.13 Definición Swagger Login

Capítulo 8 - Conclusiones y trabajo futuro

La finalidad de este Trabajo de Final de Grado ha sido la implementación de una solución a las colas de espera presenciales, utilizando una base teórica y las últimas tecnologías en Android. Nos ha servido para formarnos mejor en algunas disciplinas y para ganar experiencia en un entorno de desarrollo real.

Respecto a la implementación del proyecto se han afrontado dificultades que han llevado a revisar el alcance del proyecto y a cambiar de tecnologías una vez comenzado el desarrollo que ha llevado a replantear algunas funcionalidades que estaban dentro del planteamiento original. Sin embargo, tras estas iteraciones, el sistema queda preparado para que su implementación sea más sencilla en el futuro.

En conclusión, podemos decir que los problemas que presentaban los usuarios en las colas presenciales han sido solventados gracias a la implementación de las dos aplicaciones desarrolladas, en la que estos pueden unirse a una cola a través de un código otorgado por un administrador, ya sea numérico o por QR. Tras esto, el administrador es quien se encarga de gestionar los turnos, mientras los usuarios esperan teniendo la información relevante, como el tiempo de espera y su posición en la cola, en su dispositivo Android.

Como trabajos futuros, al nivel de diseño se podría refinar el aspecto visual de las dos aplicaciones desarrolladas. Además, convendría añadir una mayor personalización de los perfiles, dando la posibilidad de editar el perfil, incluyendo un avatar. También resultaría interesante la realización de una página web para la visualización de estadísticas sobre colas por parte de un organizador.

En lo referente a funcionalidades, el sistema está preparado para diferentes valores no utilizados, como los periodos de actividad desde la creación hasta la finalización de las colas o la ordenación de usuarios. Con esto, se podría implementar la gestión de eventos, o la posibilidad de añadir empresas a las que aportar el análisis de los datos obtenidos.

De cara a la temática del proyecto, y a las dificultades causadas por la vuelta a la nueva normalidad del COVID-19, hemos descubierto que no hay una solución implementada a estas colas presenciales que este aceptada de cara al público, lo que nos lleva a pensar que las siguientes iteraciones de este proyecto podrían ser muy beneficiosas para el uso del público general.

Chapter 8 - Conclusions and future work

The purpose of this Final Degree Project has been the implementation of a solution to face-to-face queues, using a theoretical basis and the latest technologies on Android. It has helped us to train better in some disciplines and to gain experience in a real development environment.

Regarding the implementation of the Project, we have faced difficulties that have led to review the scope of the Project and change technologies once development has begun. This has led to rethink some features that were within the original approach. However, after these iterations, the system is ready to be implemented more easily in the future.

In conclusion, we can say that the problems presented by users in the on-site queues have been solved thanks to the implementation of the two applications developed, in which users can join a queue through a code granted by an administrator, either numerical or QR code. After that, the administrator is in charge of managing the shifts, while the users wait having the relevant information, such as the waiting time and their position in the queue, in their Android device.

As future work, at the design level, the visual aspect of the two applications developed could be refined. In addition, more personalization of the profiles should be added, giving the possibility to edit the profile, including an avatar. It would also be interesting to create a website for the display of statistics on queues by an organizer.

As far as functionalities are concerned, the system is prepared to provide different unused values, such as the periods of activity during the queues lifecycle or the ordering of users. With this, it could be implemented the management of events, or the possibility of adding companies to contribute analysis of the obtained data.

In view of the subject matter of the Project, and the difficulties caused by the return to the new normality of COVID-19, we have discovered that there is no solution implemented to these queues that is accepted by the public, which leads us to think that the following iterations of this Project could be very beneficial for the use of the general public.

Trabajo conjunto e individual

Trabajo conjunto

-Definición de la BBDD (Punto 5).

-Definición de los Casos de Uso (Punto 4).

-Estado del arte (Punto 2, exceptuando el 2.3).

-Tecnologías utilizadas (Punto 3).

-Resumen y abstract.

-Conclusiones y trabajo futuro.

-Redacción de memoria.

-Anexo para la utilización de las apks.

Rubén Izquierdo

-Definición e implementación de la aplicación base de Android (Punto 6 y Anexo 2).

-Definición e implementación de la aplicación base del Servidor (Punto 7.1).

-Definición (junto a los compañeros) y soporte en la implementación de las funcionalidades de Android.

-Funcionalidad QR de Android (creación y lectura).

-Funcionalidad mostrar info/unirse cola en aplicación users.

-Funcionalidad del servidor relacionada con el Usuario.

-Funcionalidad del servidor de avanzar cola.

-Refinamiento de vistas en Android.

-Revisión de la memoria.

Daniel Piña

- Definición e implementación de la funcionalidad de *Android SignUp* con Víctor, - Crear cola, InformaciónInfoCola con Víctor y Rocío e HistorialColas junto a Rocío.

- Implementación de vistas de la aplicación Crear usuario y editar cola.

-Definición e implementación de las funcionalidades del Server 7.2.3 (*GetQueues*).

-Despliegue de Heroku junto a Víctor y Rocío.

-Creación de la base de datos en PostGreSQL junto Rocío.

-Diseño de logotipo de la aplicación.

-Definición del punto 2.1, 2.2, 5 con Rocío y 7.2 de la memoria.

-Definición de las funcionalidades del administrador con Víctor y Rocío.

-Calcular el tiempo de espera y nombre de siguiente usuario de un usuario en el *Server* y en Android.

- Refinamiento de vistas en Android.

Rocío García

-Apartado capítulo de la teoría de colas con la ayuda de las tutoras.

-Implementación de las funcionalidades *Login*, ver Información de una cola actual como ver una cola antigua en el historial esta dos últimas con Daniel y Víctor.

-Implementación de varias vistas de las aplicaciones de *User* y *Creator* en Android.

-Definición y despliegue del servidor Heroku junto a mis compañeros Daniel y Víctor.

-Creación de la base de datos en PostGreSQL junto a Daniel.

-Definición del apartado de las bases de datos con Daniel.

-Definición de las funcionalidades propias del usuario y del administrador, esta última con Víctor y Daniel.

-Swagger definiciones de *Login*, crear un usuario, unirse a una cola, crear cola *getQueues*, avanzar cola y junto a mi compañero Víctor las definiciones de parar, reanudar y cerrar cola.

-Definición e implementación parar, reanudar y cerrar cola en Android.

Víctor Gómez

-Implementación de la aplicación de Android.

-Definición e implementación de la aplicación del Servidor.

-Definición y ayuda en la implementación de las funcionalidades de Android.

-Funcionalidad del servidor relacionada con el Usuario y Colas.

-Funcionalidad de Android (Signup) junto a mi compañero Daniel, y joinQueue.

-Despliegue de Heroku junto a Daniel y a Rocío.

-Definición de las funcionalidades del administrador con Daniel y Rocío.

-Definición e implementación de unirse, parar, reanudar y cerrar cola en el servidor.

-Swagger definición de parar, reanudar y cerrar cola junto con Rocío.

Bibliografía

[1] X. A. y. X. Quesada. [En línea]. Available: https://proyectosagiles.org/que-es-scrum/.

[2] Skiplino Technologies, «https://skiplino.com/,» [En línea].

[3] QLess,Inc., «https://www.qless.com/,» [En línea].

[4] whyLine,Inc., «https://whyline.com/,» [En línea].

[5] F. J. Peláez Fermoso, J. M. Gómez García y A. García González, «Aplicaciones de la teoría de colas a la provisión óptima de servicios sociales: el caso del servicio de Teleasistencia,» Estudios de Economía Aplicada.

[6] L. A. M. S. A. F. H. Liliana Margarita Portilla, «Análisis de líneas de espera a través de teoría de colas y simulación,» *Scientia et Technica.*

[7] «kiwix.demo.ideascube.org,» [En línea]. Available: http://kiwix.demo.ideascube.org/wikipedia.es/A/Distribución_Erlang.html.

[8] http://underucaadministracionproduccion.weebly.com. [En línea]. Available: http://underucaadministracionproduccion.weebly.com/uploads/2/8/4/2/28422841/17_mc_teora_de_colas.pdf.

[9] «http://www.ub.edu/,» [En línea]. Available: http://www.ub.edu/stat/GrupsInnovacio/Statmedia/demo/Temas/Capitulo4/B0C4m1t9.htm.

[10]] «Centro de ayuda MathWorks,» [En línea]. Available: https://es.mathworks.com/help/stats/lognormal-distribution.html.

[11] «estadistica.net,» [En línea]. Available: http://www.estadistica.net/INVESTIGACION/TEORIA-COLAS.pdf.

[12] L. Cargua, «SlideShare,» [En línea]. Available: https://es.slideshare.net/LalyCargua/analisis-de-colas.

[13] R. Alvarado, «https://es.scribd.com,» [En línea]. Available: https://es.scribd.com/document/89850119/Notacion-de-Kendall.

[14] D. G. Kendall, «Stochastic process ocurring in theory queues and their analysis by the method of the imbedded Markov chain,» *The annals of mathematical satistics.*

[15] G. E. Velázquez, «https://idus.us.es/,» [En línea]. Available: https://idus.us.es/bitstream/handle/11441/77595/EstebanVel%C3%A1zquezGabriel20TFG.pdf?sequence=1.
[16] A. Ramos, «https://www.iit.comillas.edu/aramos/,» [En línea]. Available: https://www.iit.comillas.edu/aramos/simio/transpa/t_qt_ar.pdf.
[17] L. J. Rodríguez-Aragón, «previa.uclm.es,» [En línea]. Available: https://previa.uclm.es/profesorado/licesio/Docencia/mcoi/Tema3_guion.pdf.
[18] «http://virtual.umng.edu.co,» [En línea]. Available: http://virtual.umng.edu.co/distancia/ecosistema/ovas/ingenieria_civil/investigacion_de_operaciones_ii/unidad_3/DM.pdf.
[19] V. López, «Modelado Redes y AO».
[20] N. Valderrama, «https://slideplayer.es/,» [En línea]. Available: https://slideplayer.es/slide/1056487/.
[21] Google, «https://developer.android.com,» [En línea]. Available: https://developer.android.com/studio?hl=es-419.
[22] JetBrains s.r.o, «https://kotlinlang.org/,» [En línea]. Available: https://kotlinlang.org/.
[23] L. M. López, «openwebinars.net,» [En línea]. Available: https://openwebinars.net/blog/que-es-spring-framework/.
[24] Equipo Geek, «https://ifgeekthen.everis.com/,» [En línea]. Available: https://ifgeekthen.everis.com/es/spring-framework.
[25] «www.ionos.es,» [En línea]. Available: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/spring-boot-tutorial/.
[26] «www.jetbrains.com,» [En línea]. Available: https://www.jetbrains.com/es-es/datagrip/features/.
[27] L. Llamas, «www.luisllamas.es,» [En línea]. Available: https://www.luisllamas.es/realizar-peticiones-http-con-postman/.
[28] «https://www.postgresql.org,» [En línea]. Available: https://www.postgresql.org/about/.
[29] «https://es.wikipedia.org,» [En línea]. Available: https://es.wikipedia.org/wiki/Heroku.

[30] «devcenter.heroku.com,» [En línea]. Available: https://devcenter.heroku.com/articles/what-is-an-add-on.

[31] A. N. P. y. P. G. Rubén Fuentes Fernández, «Tema 03-Ingeniería de requisitos».

[32] A. Natsvlishvili, «https://medium.com/,» [En línea]. Available: https://medium.com/@annanatsv/minimum-valuable-product-mvp-a677ba7a1800.

[33] R. C. Martin, «https://es.wikipedia.org/,» [En línea]. Available: https://es.wikipedia.org/wiki/SOLID.

[34] Microsoft, «https://docs.microsoft.com,» [En línea]. Available: https://docs.microsoft.com/es-es/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

[35] R. C. M. (. Bob), «The Clean Code Blog,» 13 Agosto 2012. [En línea]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[36] «https://doc.insert-koin.io/#/,» [En línea]. Available: https://doc.insert-koin.io/#/.

[37] «Activity Life cycle of Android,» 29 Diciembre 2017. [En línea]. Available: https://medium.com/@droidbyme/activity-life-cycle-of-android-2e298809df6a.

[38] «https://developer.android.com/studio/build/build-variants,» [En línea].

[39] «https://stackoverflow.com/,» [En línea]. Available: https://stackoverflow.com/questions/415953/how-can-i-generate-an-md5-hash.

[40] U. d. Alicante, «http://www.jtech.ua.es,» [En línea]. Available: http://www.jtech.ua.es/j2ee/publico/spring-2012-13/sesion03-apuntes.html.

[41] «https://docs.spring.io/,» [En línea]. Available: https://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch02s02.html.

[42] «https://flywaydb.org/,» [En línea]. Available: https://flywaydb.org/documentation/plugins/springboot.

[43] C. B. Jurado, TDD (Test-Driven Development).

[44] «https://site.mockito.org/,» [En línea]. Available: https://site.mockito.org/.

[45] «https://junit.org/junit5/,» [En línea]. Available: https://junit.org/junit5/.

[46] F. Cejas, «https://fernandocejas.com/,» [En línea]. Available: https://fernandocejas.com/2018/05/07/architecting-android-reloaded/.

[47] «https://square.github.io/,» [En línea]. Available: https://square.github.io/retrofit/2.x/retrofit/.

[48] K. A. Wahab, «https://medium.com/,» [En línea]. Available: https://medium.com/hacktive-devs/making-network-calls-on-android-with-retrofit-kotlins-coroutines-72fd2594184b.
[49] A. Leiva, «https://antonioleiva.com/,» [En línea]. Available: https://antonioleiva.com/mvvm-vs-mvp/.
[50] «https://developer.android.com/,» [En línea]. Available: https://developer.android.com/reference/kotlin/android/content/SharedPreferences.
[51] «https://developer.android.com/,» [En línea]. Available: https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences.
[52] «Soporte de Minitab,» [En línea]. Available: https://support.minitab.com/es-mx/minitab/18/help-and-how-to/probability-distributions-and-random-data/supporting-topics/distributions/discrete-distribution/.
[53] J. P. G. Sabater, «http://personales.upv.es/,» [En línea]. Available: http://personales.upv.es/jpgarcia/linkeddocuments/teoriadecolasdoc.pdf.
[54] G. CDPYE-UGR, «www.ugr.es,» [En línea]. Available: https://www.ugr.es/~proman/Probabilidad/PDF/P_T05_TablaGeometrica.pdf.
[55] L. A. Bucki, Word 2013 Bible, John Wiley & Sons, 2013.
[56] CFI, «Cursos de Formación en Informática,» [En línea]. Available: http://cursosinformatica.ucm.es. [Último acceso: 01 06 2019].
[57] Google, «https://firebase.google.com/?hl=es,» [En línea].
[58] R. Dahl, «https://nodejs.org/es/,» [En línea].
[59] Google, «https://firebase.google.com/products/realtime-database/,» [En línea].
[60] Google, «https://firebase.google.com/products/auth/,» [En línea].
[61] Google, «https://marketingplatform.google.com/about/analytics/,» [En línea].
[62] Google, «Firebase Realtime Database - Documentación,» 03 12 2019. [En línea]. Available: https://firebase.google.com/docs/database.
[63] M. Lucila, «https://www.monografias.com/,» [En línea]. Available: https://www.monografias.com/trabajos71/teoria-colas/teoria-colas2.shtml.

[64] «Soporte de Minitab,» [En línea]. Available: https://support.minitab.com/es-mx/minitab/18/help-and-how-to/probability-distributions-and-random-data/supporting-topics/basics/continuous-and-discrete-probability-distributions/.
[65] V. L. López, Análisis de la fiabilidad.
[66] mathworks, «Centro de ayuda MathWorks,» [En línea]. Available: https://es.mathworks.com/help/stats/lognormal-distribution.html.
[67] F. Cejas. [En línea]. Available: https://fernandocejas.com/2018/05/07/architecting-android-reloaded/.
[68] «www.guru99.com,» [En línea]. Available: https://www.guru99.com/postman-tutorial.html.
[69] V. López, «Análisis de la fiabilidad».

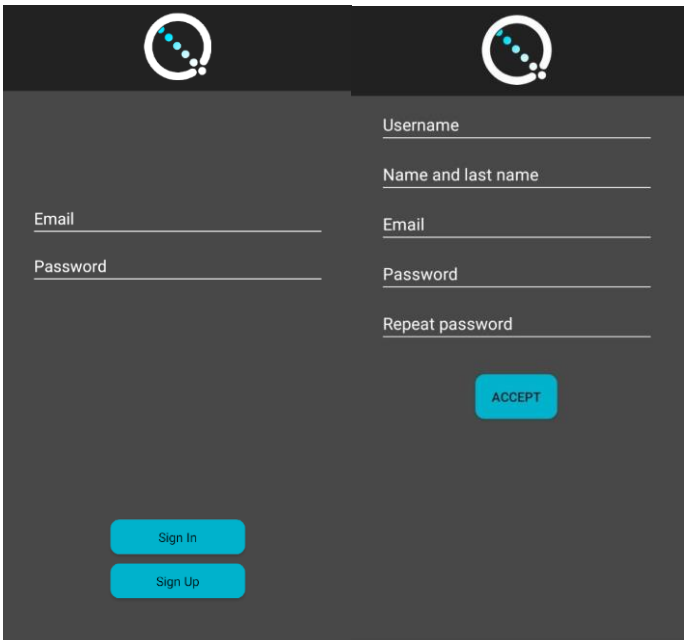
Anexos

Anexo 1

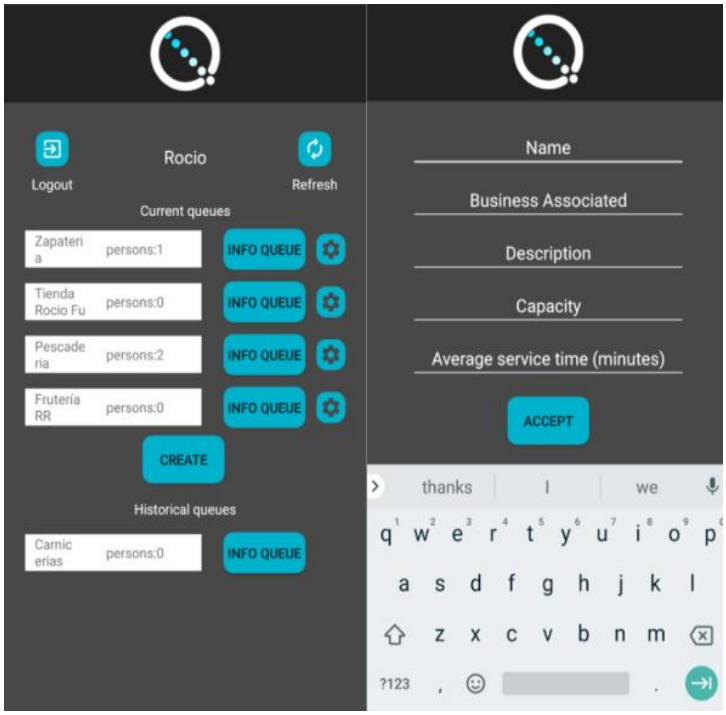
Guía para la utilización de las apks

Apk creadores

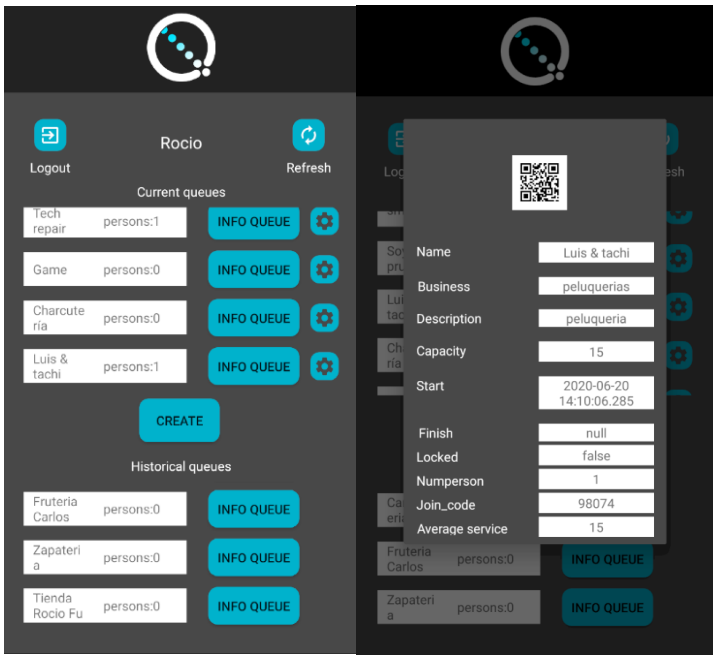
- Crear un usuario administrador



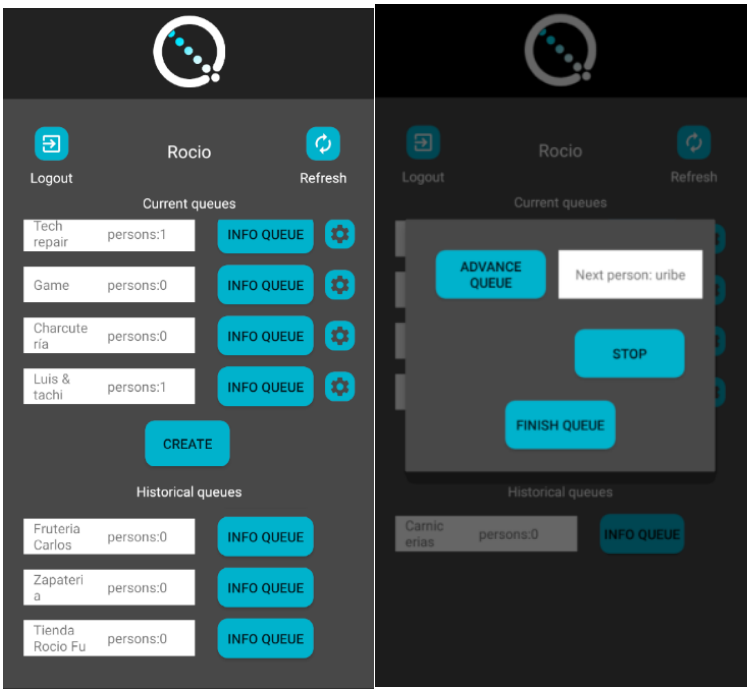
- Crear una cola con nombre, negocio, descripción, capacidad y tiempo medio estimado



- Pulsando el botón INFO, se puede ver la información asociada a la cola y el QR

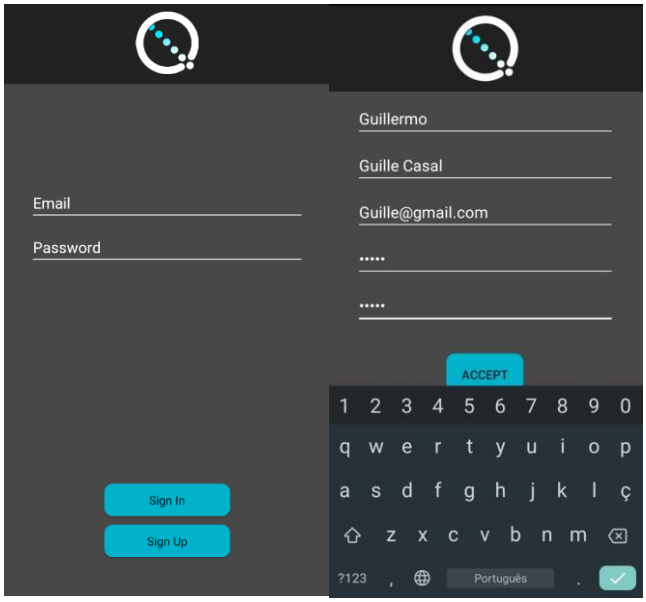


- En el botón de la tuerca lleva hacía el panel de control donde se puede avanzar, parar, reanudar y cerrar una cola, para realizar estas acciones es necesario que la cola este activa.
  - Si se cierra la cola, está se muestra en el histórico de colas.
  - Si ya no hay más usuarios en la cola o la cola esta parada, entonces la cola no puede ser avanzada



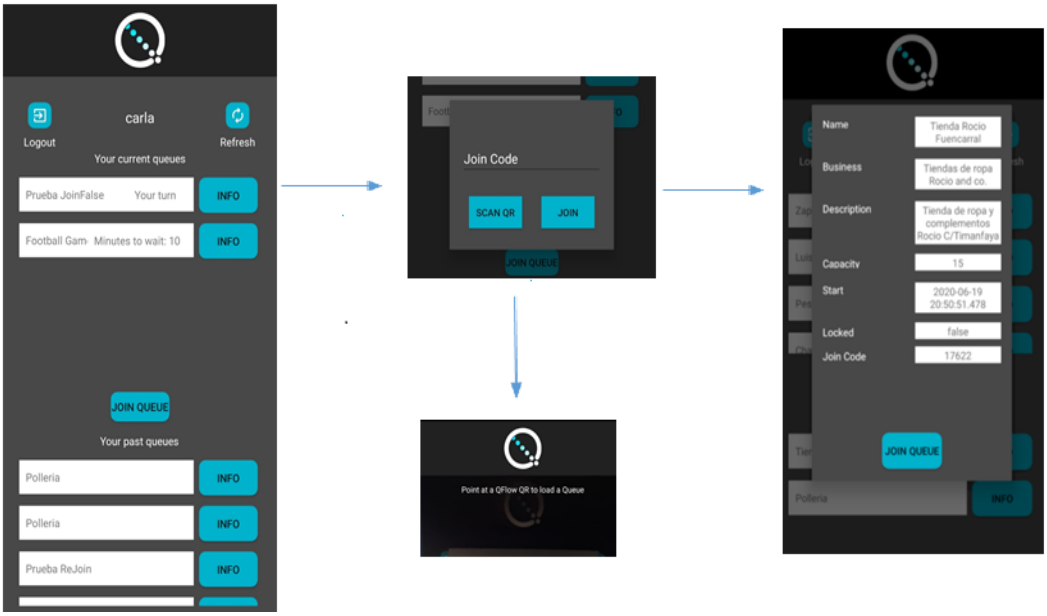
Apk usuarios

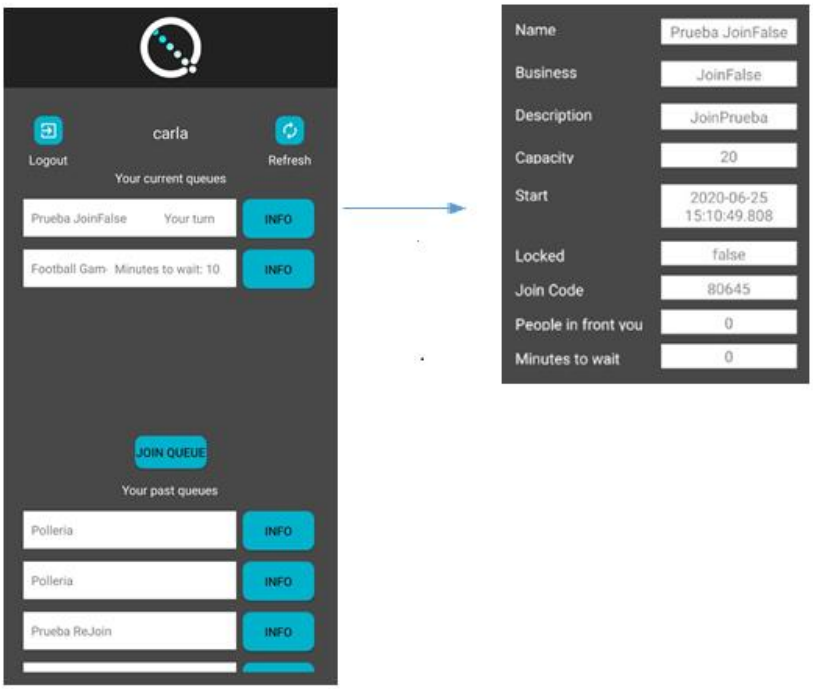
- Crear un usuario nuevo.



- En la pantalla de Home está el botón JOIN QUEUE, pulsando sobre el sale un diálogo permitiendo al usuario unirse a una cola donde puede escanear el QR de la cola, este QR hay que escanearlo de la aplicación de los creadores, o unirse a una cola mediante el código asociada a ella, hay que sacar el código(joincode)de la aplicación de los creadores, es necesario que la cola este activa para que el usuario pueda unirse.

- En el botón INFO se puede ver la información relacionada a la cola a la que se ha unido.





- Tras haber realizado la acción de unirse, aparecerá la cola a las que se ha unido el usuario apareciendo el tiempo medio estimado de servicio y si es su turno.
    - Cuando ya se ha avanzado el turno del usuario, la cola deja de aparecer en las colas actuales y pasa a estar en las colas pasadas del usuario.

Para actualizar las colas se usa el botón refresh y para salir de la sesión, logout.

Anexo 2

Arquitectura de Android

Clases importantes de la arquitectura

En el punto anterior se ha explicado la arquitectura que van a seguir las aplicaciones, mencionando sin entrar mucho en detalle sobre ciertas clases que nos van a ayudar a mantener esta arquitectura. En este apartado se expandirá sobre estas definiciones mostrando su código.

Koin

Previamente se ha comentado la necesidad de realizar la inyección de dependencias, sin embargo no se ha explicado cómo llegamos a conseguir esto.

La respuesta es Koin, una biblioteca de Kotlin que permite precisamente esto, la inyección de dependencias. Este se inicializa en nuestra clase App.kt y permite mandar instancias únicas de las clases allá donde necesitemos. Su inicialización la podemos encontrar a continuación.

```
startKoin {
    androidLogger()
    androidContext(this@App)
    modules(listOf(retrofitModule, dataModule, useCaseModule, userModule))
}
```

De tal manera que la definición de sus módulos en nuestra aplicación es la siguiente:

```
val retrofitModule = module {
    single {
        OkHttpClient.Builder()
            .connectTimeout(60, TimeUnit.SECONDS)
            .addInterceptor(HeaderInterceptor())
            .readTimeout(60, TimeUnit.SECONDS)
            .writeTimeout(60, TimeUnit.SECONDS).build()
    }

    //API Service
    single {
```

```
        Retrofit.Builder()
            .baseUrl(Constants.END_POINT_URL)
            .client(get())
            .addConverterFactory(ScalarsConverterFactory.create())
            .build().create(ApiService::class.java)
    }
}

val userModule = module {

    single<UserRepository> { UserRepository.General(get(), get()) }
    single<QueueRepository> { QueueRepository.General(get(), get(), get()) }

    single { UserAdapter }
    single { QueueAdapter }

    viewModel { LoginViewModel(get()) }
    viewModel { SignUpViewModel(get()) }
    viewModel { JoinQueueViewModel(get()) }
    viewModel { HomeViewModel(get(), get(), get())}
    viewModel { SplashScreenViewModel(get()) }
    viewModel { QRFragmentViewModel(get(), get()) }

}

val useCaseModule = module {

    factory { CreateUser(get(), get()) }
    factory { CreateAdmin(get(), get()) }
    factory { LoginCase(get(), get()) }
    factory { CreateQueue(get(), get()) }
    factory { FetchQueueById(get()) }
    factory { JoinQueue(get(), get(), get()) }
    factory { FetchQueueByJoinID(get()) }
    factory { FetchQueuesByUser(get(), get()) }
}

val dataModule = module {

    single { AppDatabase.getInstance(get()) }
```

```
    single { SharedPrefsRepository(get()) }

}
```

Either y la base de los Use Cases

Relacionado con los *Use Cases* y *ViewModel*, debemos comentar los *Eithers* [46]. Estos son una manera de aprovechar la manera que permite Kotlin tratar a las funciones, de manera que, gracias a esta clase, cuando llamemos a un Use Case, mediante un *execute*, podamos llamar a una función de éxito y otra de error, como se muestra en el siguiente ejemplo. Permitiéndonos así olvidarnos de problemas de sincronía.

```
sealed class Either<out L, out R> {
    /** * Represents the left side of [Either] class which by convention is a "Failure". */
    data class Left<out L>(val a: L) : Either<L, Nothing>()
    /** * Represents the right side of [Either] class which by convention is a "Success". */
    data class Right<out R>(val b: R) : Either<Nothing, R>()

    val isRht get() = this is Right<R>
    val isLeft get() = this is Left<L>

    fun <L> left(a: L) = Left(a)
    fun <R> right(b: R) = Right(b)

    fun either(fnL: (L) -> Any, fnR: (R) -> Any): Any =
        when (this) {
            is Left -> fnL(a)
            is Right -> fnR(b)
        }

    fun <T, L, R> Either<L, R>.flatMap(fn: (R) -> Either<L, T>): Either<L, T> =
            when (this) {
                is Left -> Left(a)
                is Right -> fn(b)
            }
}
```

Además de esta clase *Either*, se utiliza también una clase abstracta de la que heredan los Use Cases añadiendo la función *execute* [46], de manera que facilitan la

implementación de la clase anterior junto a una ejecución asíncrona. Un ejemplo de esto es:

```
abstract class UseCase<out Type, in Params, in Scope> where Type : Any, Scope :
CoroutineScope {

    abstract suspend fun run(params: Params): Either<Failure, Type>

    fun execute(onResult: (Either<Failure, Type>) -> Unit, params: Params, scope: Scope) {

        scope.launch {

            val deferred = async { run(params) }

            withContext(context = Dispatchers.Main)
            {
                onResult.invoke(deferred.await())
            }
        } class None
}
```

ApiService, el módulo Retrofit y el BaseRepository

Para las llamadas al servidor desde la aplicación se ha utilizado la librería de *Retrofit* [47], una biblioteca de Android creada con esta finalidad. La de realizar y facilitar las llamadas a Apis desde aplicaciones en Java.

Todas las llamadas realizadas con *Retrofit*, quedan definidas en nuestra interfaz ApiService a la cual llamaremos desde los repositorios. El resultado final sería uno como se muestra en el ejemplo.

```
interface ApiService {

    val prefs : SharedPrefsRepository
    companion object Factory {

        //Headers & Params
        const val PARAM_QUEUE_ID = "idQueue"
        const val HEADER_IS_ADMIN = "isAdmin"
        const val HEADER_EMAIL = "mail"
        const val HEADER_PASS = "password"
        const val HEADER_TOKEN = "token"
        const val PARAM_JOIN_ID = "joinId"
        const val PARAM_FINISHED = "finished"
        const val PARAM_EXPAND = "expand"



        //URL
        const val POST_JOIN_QUEUE = "qflow/queues/joinQueue/{$PARAM_JOIN_ID}"
        const val POST_CREATE_QUEUE = "qflow/queues/"
        const val GET_QUEUE_USERID = "qflow/queues/byIdUser/"
        const val GET_QUEUE_QUEUEID = "qflow/queues/byIdQueue/"
        const val GET_QUEUE_JOINID = "qflow/queues/byIdJoin/{$PARAM_JOIN_ID}"
        const val POST_CREATE_USER = "qflow/user/"
        const val PUT_LOGIN_USER = "qflow/user/"

    }
    @Headers("Content-type: application/json")
    @GET(GET_QUEUE_USERID)
    fun getQueuesByUser(
        @Header(HEADER_TOKEN) token: String,
        @Query(PARAM_EXPAND) expand: String?,
        @Query(PARAM_FINISHED) finished: Boolean?
    ) : Call<String>

    @GET(GET_QUEUE_QUEUEID)
    fun getQueueByQueueId(@Path(PARAM_QUEUE_ID) idQueue: Int): Call<String>

    @Headers("Content-type: application/json")
    @POST(POST_CREATE_USER)
    fun postCreateUser(@Body body: String, @Header(HEADER_IS_ADMIN) admin: Boolean):
```

```
Call<String>

    @Headers("Content-type: application/json")
    @PUT(PUT_LOGIN_USER)
    fun postLoginUser(
        @Header(HEADER_IS_ADMIN) admin: Boolean,
        @Header(HEADER_EMAIL) email: String,
        @Header(HEADER_PASS) password:String
    ): Call<String>

    @Headers("Content-type: application/json")
    @POST(POST_CREATE_QUEUE)
    fun postQueue(@Body body: String,
                  @Header(HEADER_TOKEN) token: String
    ): Call<String>

    @Headers("Content-type: application/json")
    @POST(POST_JOIN_QUEUE)
    fun postJoinQueue(@Path(PARAM_JOIN_ID) joinId: Int,
                      @Header(HEADER_TOKEN) token: String): Call<String>

    @Headers("Content-type: application/json")
    @GET(GET_QUEUE_JOINID)
    fun getQueueByJoinId(@Path(PARAM_JOIN_ID) idJoin: Int): Call<String>
}
```

Además de esta interfaz, también hay una clase llamada *HeaderInterceptor* que se encarga de interceptar el *request* y añadirle los *headers* que necesitemos. En nuestro caso, aprovechamos este interceptor para enviar un token de usuario siempre que se necesita hacer alguna acción con las colas.

Finalmente, en cuanto a la realización de *requests* de la aplicación, falta por mencionar como se realizan estas llamadas desde cada uno de los repositorios. Para ello, se ha definido una clase BaseRepository [48], como se muestra a continuación, la cual define una función base a utilizar en las peticiones al servicio

```
abstract class BaseRepository
{
   fun <T, R> request(call: Call<T>, transform: (T) -> R, default: T): Either<Failure,R> {
      return try {
         val response = call.execute()
         when (response.isSuccessful) {
            true -> Either.Right(transform((response.body() ?: default)))
            false -> {
               Either.Left(Failure.ServerErrorCode(response.code()))
            }
         }
      } catch (exception: Throwable) {
         Either.Left(Failure.ServerException(exception))
      }
   }
}
```

En esta clase se encuentran dos funciones, una para hacer peticiones a *Firebase*, la cual se utilizaba antes de cambiar el servicio por uno propio, y otra que es la que se utiliza para las peticiones a través de *Retrofit*.

Como se puede apreciar en el ejemplo anterior esta recibe por el primer parámetro la función que vamos a ejecutar, seguido de la respuesta que queremos devolver en el tipo que sea requerido. Sin embargo, lo más importante de estas funciones es que respetan la implementación de los *Eithers* comentada anteriormente.

ScreenStates

En los dos últimos apartados se ha tratado el cómo un *Use Case* se ejecuta desde el *ViewModel* y cómo se realizan las llamadas al servicio para realizar aquello que el *Use Case* necesite. Ahora vamos a explicar cómo tras realizarse esa ejecución actualizamos la vista para mostrar sus resultados.

Para ello, se usa una instancia observable de los *ScreenStates* [49]. Estos representan los posibles estados en los que se puede encontrar una vista. De manera que tras realizar un Use Case, actualizamos el valor de estos para que representen correctamente el estado al que se ha llevado la vista. A continuación, se muestra la clase base del *ScreenState*.

```
sealed class ScreenState<out T>
{
   object Loading: ScreenState<Nothing>()
   class Render<T>(val renderState: T) : ScreenState<T>()
}
```

Esta clase, como podemos observar en el siguiente ejemplo, determina que una vista puede tener dos estados básicos, una de carga, en la cual cargaremos el *loader*, y otra de renderizado, en la que se aplicarán los resultados obtenidos en una ejecución en el *ViewModel*.

```
sealed class SplashScreenScreenState {

   object UserIsLogged : SplashScreenScreenState()
   object UserIsNotLogged : SplashScreenScreenState()

}
```

```
class SplashScreenViewModel(private val sharedPrefsRepository: SharedPrefsRepository) :
    BaseViewModel(), KoinComponent {

    private val _screenState: MutableLiveData<ScreenState<SplashScreenScreenState>> =
        MutableLiveData()
    val screenState: LiveData<ScreenState<SplashScreenScreenState>>
        get() = _screenState

    fun checkIfUserIsLogged() {
        if (sharedPrefsRepository.getUserToken() != null)
            _screenState.value = ScreenState.Render(SplashScreenScreenState.UserIsLogged)
        else
            _screenState.value = ScreenState.Render(SplashScreenScreenState.UserIsNotLogged)
    }

}
```

Un ejemplo lo podríamos ver en los estados que tiene la vista de entrada de la aplicación en la cual se realiza una llamada al *ViewModel* para que compruebe si un usuario se ha logueado anteriormente. Una vez se comprueba, se actualizará el valor de un objeto estado, que se encuentra en el *ViewModel* y es observado por la vista, de manera que represente si hay un usuario o no. Así, al cambiar el valor de este objeto, la vista podrá realizar una u otra funcionalidad.

Shared Preferences Repository

Anteriormente durante la explicación del *BaseRepository*, se ha mencionado que se manda un *token* de usuario junto a los *requests*. Este *token* también se utiliza en la vista de entrada mencionada en el ejemplo del apartado anterior para comprobar si un usuario se ha logueado anteriormente.

Dicho *token* se obtiene tanto de la *request* del *Login*, cómo del *SignUp*. Una vez obtenido, existe la necesidad de guardarlo en la aplicación para que, después de cerrarla, este *token* siga ahí. Sin embargo, no se puede guardar a la ligera, ya que éste permite el acceso a todos los datos del usuario en el servidor,

Para ello se ha utilizado una extensión del *SharedPreferences* [50] de Android, conocida como *EncryptedSharedPreferences* [51], una biblioteca relativamente nueva que junto a la funcionalidad de la biblioteca base añade una capa de seguridad extra aplicando claves y firmas. La implementación la podemos encontrar a continuación:

```
class SharedPrefsRepository(c : Context) {

    companion object{
        const val ID_USER = "ID_USER"
    }

    private val prefs : SharedPreferences = c.getSharedPreferences("prefs",
MODE_PRIVATE)
    private val encryptedShared: SharedPreferences



    init {
        val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)

        encryptedShared = EncryptedSharedPreferences.create(
            "secret_shared_prefs",
            masterKeyAlias,
            c,
            EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
            EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
        )

    }

    fun putUserToken(token : String){
        encryptedShared.edit().putString(ID_USER, token).apply()
    }

    fun removeUserToken(){
        encryptedShared.edit().remove(ID_USER).apply()
    }

    fun getUserToken() : String?{
        return (encryptedShared.getString(ID_USER, null))
```

```
}

fun clearNormal(){
    prefs.edit().clear().apply()
}

fun clearEncrypter(){
    encryptedShared.edit().clear().apply()
}
```

Anexo 3

Despliegue de Heroku

Para poder desplegar el servidor en Heroku usamos las herramientas de Git, GitHub y Heroku CLI.

Para ello es necesario inicializar un repositorio local de Git y confirmar el código de la aplicación. Se hace a través los siguientes comandos:

```
$ cd myapp
$ git-init
$ git add
$ git commit -m “My first commit”
```

Después, es necesario subir la aplicación a Heroku a través de la consola de Heroku CLI mediante los siguientes comandos:

- Creación de una aplicación de Heroku vacía:

  ```
  $ heroku create.
  ```

- Comprobación de creación correcta de la aplicación:

  ```
  $ git remote -v
  ```

- Subida del código de nuestro servidor a Heroku a través de la rama *master* de nuestro repositorio:

  ```
  $ git push heroku master
  ```

Para sincronizar Heroku con Github se realiza el método de *deploy* mediante el panel de control de Heroku seleccionando, además de añadir la opción de despliegues automáticos desde la rama *master*.

Anexo 4

Distribuciones de tipo discreto

Una distribución discreta [52] describe la probabilidad de que ocurran eventos discretos, es decir, es una lista de los diferentes valores de interés y sus probabilidades asociadas.

Estas distribuciones son útiles en la teoría de colas para calcular el número de clientes en un intervalo de tiempo [53].

Bernoulli: Es una distribución de probabilidad discreta que solo puede tomar dos valores, con una probabilidad p de que ocurra un suceso y una probabilidad de q= 1 -p de que ocurra el suceso contrario.

Binomial: Ejecuta n veces un experimento de Bernoulli, las variables que definen al proceso son: la cantidad de veces que se ejecuta (n) la probabilidad de éxito(p), la probabilidad de fracaso (q=p-1) y las veces que se obtiene el éxito en las veces que se ejecuta(k).

Geométrica: Se aplica a una secuencia de experimentos [54] de Bernoulli independientes con un evento de interés que tiene una probabilidad p, hasta que aparece el primer éxito y se mide el número de fracasos.

Binomial negativa: modela el número de ensayos necesarios para producir un numero especifico de eventos.